
UvA DL Notebooks

Release v1.2

Phillip Lippe

Nov 26, 2022

GUIDES

1 Schedule (Deep Learning 1, edition 2022)	3
2 How to run the notebooks	5
3 Tutorial-Lecture alignment	7
4 Feedback, Questions or Contributions	9
4.1 Guide 1: Working with the Lisa cluster	9
4.2 Guide 2: Research projects with PyTorch	20
4.3 Guide 3: Debugging in PyTorch	24
4.4 Guide 4: Research Projects with JAX	30
4.5 Tutorial 2: Introduction to PyTorch	59
4.6 Tutorial 3: Activation Functions	83
4.7 Tutorial 4: Optimization and Initialization	105
4.8 Tutorial 5: Inception, ResNet and DenseNet	133
4.9 Tutorial 6: Transformers and Multi-Head Attention	159
4.10 Tutorial 7: Graph Neural Networks	193
4.11 Tutorial 8: Deep Energy-Based Generative Models	215
4.12 Tutorial 9: Deep Autoencoders	233
4.13 Tutorial 10: Adversarial attacks	254
4.14 Tutorial 11: Normalizing Flows for image modeling	274
4.15 Tutorial 12: Autoregressive Image Modeling	306
4.16 Tutorial 15: Vision Transformers	333
4.17 Tutorial 16: Meta-Learning - Learning to Learn	345
4.18 Tutorial 17: Self-Supervised Contrastive Learning with SimCLR	376
4.19 Tutorial 2 (JAX): Introduction to JAX+Flax	395
4.20 Tutorial 3 (JAX): Activation Functions	422
4.21 Tutorial 4 (JAX): Optimization and Initialization	445
4.22 Tutorial 5 (JAX): Inception, ResNet and DenseNet	474
4.23 Tutorial 6 (JAX): Transformers and Multi-Head Attention	498
4.24 Tutorial 7 (JAX): Graph Neural Networks	533
4.25 Tutorial 9 (JAX): Deep Autoencoders	543
4.26 Tutorial 11 (JAX): Normalizing Flows for image modeling	565
4.27 Tutorial 12 (JAX): Autoregressive Image Modeling	597
4.28 Tutorial 15 (JAX): Vision Transformers	628
4.29 Tutorial 17 (JAX): Self-Supervised Contrastive Learning with SimCLR	643
4.30 GDL - Regular Group Convolutions	670
4.31 GDL - Steerable CNNs	714
4.32 DPM1 - Deep Probabilistic Models I	766
4.33 DPM2 - Variational inference for deep discrete latent variable models	815

4.34	DPM 2 - Variational Inference for Deep Continuous LVMs	860
4.35	AGM - Advanced Topics in Normalizing Flows - 1x1 convolution	903
4.36	HDL - Introduction to HyperParameter Tuning	908
4.37	HDL - Introduction to Multi GPU Programming	912
4.38	BNN - Comparing to Non-Bayesian Methods for Uncertainty	929
4.39	DNN - Tutorial 2 Part I: Physics inspired Machine Learning	955
4.40	DNN - Tutorial 2 Part II: Physics inspired Machine Learning	968
4.41	DS - Dynamical Systems & Neural ODEs	996
4.42	SGA - Sampling Discrete Structures	1033
4.43	SGA - Sampling Subsets with Gumbel-Top k Relaxations	1041
4.44	SGA: Learning Latent Permutations with Gumbel-Sinkhorn Networks	1053
4.45	SGA - Graph Sampling for Neural Relational Inference	1070

Course website: <https://uvadlc.github.io/>

Course edition: DL1 - Fall 2022, DL2 - Spring 2022, Being kept up to date

Repository: https://github.com/phlippe/uvadlc_notebooks

Recordings: YouTube Playlist

Author: Phillip Lippe

Note: Interested in learning JAX? We have recently started translating the notebooks from PyTorch to JAX+Flax. Check out our new notebooks in the tab *Deep Learning I (JAX+Flax)* to learn how you can speedup the model training with JAX!

For this year's course edition, we created a series of Jupyter notebooks that are designed to help you understanding the "theory" from the lectures by seeing corresponding implementations. We will visit various topics such as optimization techniques, transformers, graph neural networks, and more (for a full list, see below). The notebooks are there to help you understand the material and teach you details of the PyTorch framework, including PyTorch Lightning. Further, we provide one-to-one translations of the notebooks to JAX+Flax as alternative framework.

The notebooks are presented in the first hour of every group tutorial session. During the tutorial sessions, we will present the content and explain the implementation of the notebooks. You can decide yourself whether you just want to look at the filled notebook, want to try it yourself, or code along during the practical session. The notebooks are not directly part of any mandatory assignments on which you would be graded or similarly. However, we encourage you to get familiar with the notebooks and experiment or extend them yourself. Further, the content presented will be relevant for the graded assignment and exam.

The tutorials have been integrated as official tutorials of PyTorch Lightning. Thus, you can also view them in [their documentation](#).

**CHAPTER
ONE**

SCHEDULE (DEEP LEARNING 1, EDITION 2022)

Date	Notebook
Tuesday, 1. November 2022, 17.00-18.00	Tutorial 2: Introduction to PyTorch
Tuesday, 8. November 2022, 17.00-18.00	Tutorial 3: Activation functions
Tuesday, 15. November 2022, 17.00-18.00	Tutorial 4: Optimization and Initialization
Tuesday, 22. November 2022, 17.00-18.00	Tutorial 5: Inception, ResNet and DenseNet
Tuesday, 29. November 2022, 17.00-18.00	Tutorial 6: Transformers and Multi-Head Attention
Tuesday, 6. December 2022, 17.00-18.00	Tutorial 7: Graph Neural Networks
Tuesday, 13. December 2022, 17.00-18.00	Tutorial 17: Self-Supervised Contrastive Learning with SimCLR

CHAPTER
TWO

HOW TO RUN THE NOTEBOOKS

On this website, you will find the notebooks exported into a HTML format so that you can read them from whatever device you prefer. However, we suggest that you also give them a try and run them yourself. There are three main ways of running the notebooks we recommend:

- **Locally on CPU:** All notebooks are stored on the github repository that also builds this website. You can find them here: https://github.com/phlippe/uadlc_notebooks/tree/master/docs/tutorial_notebooks. The notebooks are designed so that you can execute them on common laptops without the necessity of a GPU. We provide pre-trained models that are automatically downloaded when running the notebooks, or can manually be downloaded from this [Google Drive](#). The required disk space for the pretrained models and datasets is less than 1GB. To ensure that you have all the right python packages installed, we provide a conda environment in the [same repository](#) (choose the CPU or GPU version depending on your system).
- **Google Colab:** If you prefer to run the notebooks on a different platform than your own computer, or want to experiment with GPU support, we recommend using [Google Colab](#). Each notebook on this documentation website has a badge with a link to open it on Google Colab. Remember to enable GPU support before running the notebook (`Runtime -> Change runtime type`). Each notebook can be executed independently, and doesn't require you to connect your Google Drive or similar. However, when closing the session, changes might be lost if you don't save it to your local computer or have copied the notebook to your Google Drive beforehand.
- **Lisa cluster:** If you want to train your own (larger) neural networks based on the notebooks, you can make use of the Lisa cluster. However, this is only suggested if you really want to train a new model, and use the other two options to go through the discussion and analysis of the models. Lisa might not allow you with your student account to run Jupyter notebooks directly on the gpu_shared partition. Instead, you can first convert the notebooks to a script using `jupyter nbconvert --to script ...ipynb`, and then start a job on Lisa for running the script. A few advices when running on Lisa:
 - Disable the tqdm statements in the notebook. Otherwise your slurm output file might overflow and be several MB large. In PyTorch Lightning, you can do this by setting `enable_progress_bar=False` in the trainer.
 - Comment out the matplotlib plotting statements, or change `plt.show()` to `plt.savefig(...)`.

**CHAPTER
THREE**

TUTORIAL-LECTURE ALIGNMENT

We will discuss 7 of the tutorials in the course, spread across lectures to cover something from every area. You can align the tutorials with the lectures based on their topics. The list of tutorials in the Deep Learning 1 course is:

- Guide 1: Working with the Lisa cluster
- Tutorial 2: Introduction to PyTorch
- Tutorial 3: Activation functions
- Tutorial 4: Optimization and Initialization
- Tutorial 5: Inception, ResNet and DenseNet
- Tutorial 6: Transformers and Multi-Head Attention
- Tutorial 7: Graph Neural Networks
- Tutorial 8: Deep Energy Models
- Tutorial 9: Autoencoders
- Tutorial 10: Adversarial attacks
- Tutorial 11: Normalizing Flows on image modeling
- Tutorial 12: Autoregressive Image Modeling
- Tutorial 15: Vision Transformers
- Tutorial 16: Meta Learning - Learning to Learn
- Tutorial 17: Self-Supervised Contrastive Learning with SimCLR

CHAPTER
FOUR

FEEDBACK, QUESTIONS OR CONTRIBUTIONS

This is the first time we present these tutorials during the Deep Learning course. As with any other project, small bugs and issues are expected. We appreciate any feedback from students, whether it is about a spelling mistake, implementation bug, or suggestions for improvements/additions to the notebooks. Please use the following [link](#) to submit feedback, or feel free to reach out to me directly per mail (p dot lippe at uva dot nl), or grab me during any TA session.

If you find the tutorials helpful and would like to cite them, you can use the following bibtex:

```
@misc{lippe2022uvadlc,
    title      = {{UvA Deep Learning Tutorials}},
    author     = {Phillip Lippe},
    year       = 2022,
    howpublished = {\url{https://uvadlc-notebooks.readthedocs.io/en/latest/}}
}
```

4.1 Guide 1: Working with the Lisa cluster

Author: Phillip Lippe

This tutorial explains how to work with the Lisa cluster for the Deep Learning course at the University of Amsterdam. Every student will receive an account to have resources for training deep neural networks and get familiar with working on a cluster. It is recommended to have listened to the presentation by the SURFsara team or the TA team before going through this tutorial. Further, this tutorial assumes that you are familiar with using the terminal in Linux. If not, a crash course can be found [here](#).

4.1.1 The Lisa cluster

The following section gives a quick introduction to the Lisa cluster, and how it is build up. A detailed description of the system can be found in the SURFsara [user guide](#).

What is a cluster computer?

(Disclaimer: the following paragraph is an adapted version of the [Lisa user guide](#). Credits: SURFsara team)

You can imagine a cluster computer as a collection of regular computers (known as nodes), tied together with network cables that are similar to the network cables in your home or office (see the figure below - credit: SURFsara team). Each node has its own CPU, memory and disk space, in addition to which they generally have access to a shared file system. On a cluster computer, you can run hundreds of computational tasks simultaneously.

Interacting with a cluster computer is different from a normal computer. Normal computers are mostly used interactively, i.e. you type a command or click with your mouse, and your computer instantly responds by e.g. running a program. Cluster computers are mostly used non-interactively.

A cluster computer such as Lisa mainly has two types of nodes: login nodes and batch nodes. You connect to Lisa through the login node (see next section). This is an interactive node: similar to your own PC, it immediately responds to the commands you type. There are only a few login nodes on a cluster computer, and you only use them for light tasks: adjusting your code, preparing your input data, writing job scripts, etc. Since the login nodes are only meant for light tasks, many users can be on the same login node at the same time. To prevent users from over-using the login node, any command that takes longer than 15 minutes will be killed.

Your ‘big’ calculations such as a neural network training will be done on the batch nodes. These perform what is known as batch jobs. A batch job is essentially a recipe of commands (put together in a job script) that you want the computer to execute. Calculations on the batch nodes are not performed right away. Instead, you submit your job script to the job queue. As soon as sufficient resources (i.e. batch nodes) are available for your job, the system will take your job from the queue, and send it to the batch nodes for execution.

Architecture of Lisa

A visual description of the Lisa architecture can be found below (figure credit - SURFsara team). You can connect to any login node of Lisa to interact with the system. Over the login nodes, you have access to the shared file system across nodes. The one you will mainly interact with is `/home` where you can store your code, data, etc. You can access your files from any login node, as well as any compute node. You have a maximum disk space of 200GB which should be sufficient for the DL course.

You do not directly interact with any compute node. Instead, you can request computational resources with a job script, and Lisa will assign a compute node to this job using a [SLURM job scheduler](#). If all computational resources are occupied, your job will be placed in a queue, and scheduled when resources are available.

Lisa has multiple sets of compute nodes with different computational resources, also called partitions. The one we will use for the Deep Learning course is called `gpu_shared_course`, and provides us compute nodes with the following resources:

Processor	CPU Cores	RAM Memory	GPUs
Bronze 3104 (1.7GHz)	12	256GB	4x GeForce 1080Ti, 11 GB GDDR5x

These computational resources are more than sufficient for the assignments in this course. For a job, we usually only use a single GPU from a compute node, meaning that we would also use 1/4th of the other resources (3 CPU cores, 64GB RAM). The scheduler of Lisa will assign multiple jobs to the same node if its computational resources are not exhausted yet, thus not wasting any if we only use a single GPU.

4.1.2 First steps

After discussing the general architecture of Lisa, we are ready to discuss the practical aspects of how to use the Lisa cluster.

REMINDER: When you first receive your login data for Lisa, make sure to go to the [user portal](#) and change the password.

How to connect to Lisa

You can login to Lisa's login nodes using a secure shell (SSH):

```
ssh -X lcur____@lisa.surfsara.nl
```

Replace `lcur____` by your username. You will be connected to one of its login nodes, and have the view of a standard Linux system in your home directory. Note that you should only use the login node as an interface, and not as compute unit. Do not run any trainings on this node, as it will be killed after 15 minutes, and slows down the communication with Lisa for everyone. Instead, Lisa uses a SLURM scheduler to handle computational expensive jobs (see below).

If you want to transfer files between Lisa and your local computer, you can use standard Unix commands such as `scp` or `rsync`, or graphical interfaces such as [FileZilla](#) (use port 22 in FileZilla) or [WinSCP](#) (for Windows PC). A copy operation from Lisa to your local computer with `rsync`, started from your local computer, could look as follows:

```
rsync -av lcur____@lisa.surfsara.nl:~/source destination
```

Replace `lcur____` by your username, `source` by the directory/file on Lisa you want to copy on your local machine, and `destination` by the directory/file it should be copied to. Note that `source` is referenced from your home directory on Lisa. If you want to copy a file from your local computer to Lisa, use:

```
rsync -av source lcur____@lisa.surfsara.nl:~/destination
```

Again, replace `source` with the directory/file on your local computer you want to copy to Lisa, and `destination` by the directory/file it should be copied to.

Modules

Lisa uses modules to provide you various pre-installed software. This includes simple Python, but also the NVIDIA libraries CUDA and cuDNN that can be necessary to access GPUs. However, for our course, we only need the Anaconda module:

```
module load 2021
module load Anaconda3/2021.05
```

Note that there also exist a 2022 module with slightly newer package versions, but it is at the moment not functional. Hence, we stick with the 2021 module. The CUDA and cuDNN libraries are already taken care of by installing the cudatoolkits in conda.

Install the environment

To run the Deep Learning assignments and other code like the notebooks on Lisa, you need to install the provided environment for Lisa (dl2022_gpu.yml). You can either download it locally and copy it to your Lisa account via rsync or scp as described before, or simply clone the [practicals github](#) on Lisa:

```
git clone https://github.com/uvadlc/uvaldc_practicals_2022.git
```

Lisa provides an Anaconda module, which you can load via `module load Anaconda3/2021.05` as mentioned before (remember to load the 2021 module beforehand). We recommend installing the package via a job file since the installation can take 20-30 minutes, and any command on the login node will be killed without warning after 15 minutes.

To do that, save the following content into a file called `install_environment.job` in your home directory:

```
#!/bin/bash

#SBATCH --partition=gpu_shared_course
#SBATCH --gres=gpu:0
#SBATCH --job-name=InstallEnvironment
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=3
#SBATCH --time=04:00:00
#SBATCH --mem=32000M
#SBATCH --output=slurm_output_%A.out

module purge
module load 2021
module load Anaconda3/2021.05

cd $HOME/uvadlc_practicals_2022/
conda env create -f dl2022_gpu.yml
```

You can use e.g. `nano` to do that. If the environment file is not in the cloned repository or you haven't cloned the repo, change the `cd` statement to the directory where it is stored. Once the file is saved, start the job with the command `sbatch install_environment.job`. The installation process is started on a compute node with a time limit of 4 hours, which should be sufficiently long. Let's look at the next section to understand what we have actually done here with respect to 'job files'.

Trouble shooting

If the installation via job file does not work, try to install the environment with the following command from the login node after navigating to the directory the environment file is in:

```
conda env create -f dl2022_gpu.yml
```

Note that the jobs on the login node on Lisa are limited to 15 minutes. This is often not enough to install the full environment. If the installation command is killed, you can simply restart it. If you get the error that a package is corrupted, go to `/home/lcur____/.conda/pkgs/` under your home directory and remove the directory of the corrupted package. If you get the error that the environment `dl2022` already exists, go to `/home/lcur____/.conda/envs/`, and remove the folder '`dl2022`'.

If you experience issues with the Anaconda module, you can also install Anaconda yourself ([download link](#)) or ask your TA for help.

Verifying the installation

When the installation process is completed, you can check if the process was successful by activating your environment on the login node via `source activate dl2022` (remember to have loaded the anaconda module beforehand), and starting a python console with executing `python`. It should say `Python 3.10.6 | packaged by conda-forge`. If you see a different python version, you might not have activated the environment correctly.

In the python console, try to import pytorch via `import torch` and check the version: `torch.__version__`. It should say `1.13.0`. Finally, check whether PyTorch can access the GPU: `torch.cuda.is_available()`. Note that in most cases, this will return `False` because most login-nodes on Lisa do not have GPUs. You can login to a GPU node via `ssh lcur...@login-gpu.lisa.surfsara.nl`, and on this node, you should see that the command returns `True`. If that is the case, you should be all set.

4.1.3 The SLURM scheduler

Lisa relies on a SLURM scheduler to organize the jobs on the cluster. When logging into Lisa, you cannot just start a python script with your training, but instead submit a job to the scheduler. The scheduler will decide when and on which node to run your job, based on the number of nodes available and other jobs submitted.

Job files

We provide a template for a job file that you can use on Lisa. Create a file with any name you like, for example `template.job`, and start the job by executing the command `sbatch template.job`.

```
#!/bin/bash

#SBATCH --partition=gpu_shared_course
#SBATCH --gres=gpu:1
#SBATCH --job-name=ExampleJob
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=3
#SBATCH --time=04:00:00
#SBATCH --mem=32000M
#SBATCH --output=slurm_output_%A.out

module purge
module load 2021
module load Anaconda3/2021.05

# Your job starts in the directory where you call sbatch
cd $HOME/...
# Activate your environment
source activate dl2022
# Run your code
srun python -u ...
```

Job arguments

You might have to change the #SBATCH arguments depending on your needs. We describe the arguments below:

- **partition**: The partition of Lisa on which you want to run your job. As a student, you only have access to the partition gpu_shared_course, which provides you nodes with NVIDIA GTX1080Ti GPUs (11GB).
- **gres**: Generic resources include the GPU which is crucial for deep learning jobs. You can select up to two GPUs with your account, but if you haven't designed your code to explicitly run on multiple GPUs, please use only one GPU (so no need to change what we have above).
- **job-name**: Name of the job to pop up when you list your jobs with squeue (see below).
- **ntasks**: Number of tasks to run with the job. In our case, we will always use 1 task.
- **cpus-per-task**: Number of CPUs you request from the nodes. The gpu_shared_course partition restricts you to max. 3 CPUs per job/GPU.
- **time**: Estimated time your job needs to finish. It is no problem if your job finishes earlier than the specified time. However, if your job takes longer, it will be instantaneously killed after the specified time. Still, don't specify unnecessarily long times as this causes your job to be scheduled later (you need to wait longer in the queue if other people also want to use the cluster). A good rule of thumb is to specify ~20% more than what you would expect.
- **mem**: RAM of the node you need. Note that this is *not* the GPU memory, but the random access memory of the node. On gpu_shared_course, you are restricted to 64GB per job/GPU which is more than you need for the assignments.
- **output**: Output file to which the slurm output should be written. The tag "%A" is automatically replaced by the job ID. Note that if you specify the output file to be in a directory that does not exist, no output file will be created.

SLURM allows you to specify many more arguments, but the ones above are the important ones for us. If you are interested in a full list, see [here](#).

Scratch

If you work with a lot of data, or a larger dataset, it is advised to copy your data to the /scratch directory of the node. Otherwise, the read/write operation might become a bottleneck of your job. To do this, simply use your copy operation of choice (`cp`, `rsync`, ...), and copy the data to the directory `$TMPDIR`. You should add this command to your job file before calling `srun` Remember to point to this data when you are running your code. If you have a dataset that can be downloaded, you can also directly download it to the scratch (can sometimes be faster than copying). In case you also write something on the scratch, you need to copy it back to your home directory before finishing the job.

Edit Dec. 6, 2021: Due to internal changes to the filesystem of Lisa, it is required to **use the scratch for any dataset** such as CIFAR10. In PyTorch, the CIFAR10 dataset is structured into multiple large batches (usually 5), and only that batch is loaded which is currently needed. This is why during training, it requires a lot of reading operations on the disk which can slow down your training and constitutes a challenge to the Lisa system when hundreds of students share the same filesystem. Hence, you have to use the scratch for such datasets. For most parts in the assignment, you can do this by specifying `--data_dir $TMPDIR` on the python command in your job file (check if the argument parser has this argument, otherwise you can add it yourself). This will download the dataset to the scratch and only load it from there. We recommend using this approach also for future course editions.

Starting and organizing jobs

To start a job, you simply have to run `sbatch jobfile` where you replace `jobfile` by the filename of the job. Note that no specific file postfix like `.job` is necessary for the job (you can use `.txt` or any other you prefer). After your job has been submitted, it will be first placed into a waiting queue. The SLURM scheduler decides when to start your job based on the time of your job, all other jobs currently running or waiting, and available nodes.

Besides `sbatch`, you can interact with the SLURM scheduler via the following commands:

- `squeue`: Lists all jobs that are currently submitted to Lisa. This can be a lot of jobs as it includes all partitions. You can make it partition-specific using `squeue -p gpu_shared_course`, or only list the jobs of your account: `squeue -u lcur____` (again, replace `lcur____` by your username). See the [slurm documentation](#) for details.
- `scancel JOBID`: Cancels and stops a job, independent of whether it is running or pending. The job ID can be found using `squeue`, and is printed when submitting the job via `sbatch`.
- `scontrol show job JOBID`: Shows additional information of a specific job, like the estimated start time.

Interactive sessions

As alternative to running jobs via `sbatch`, you can request an interactive job session. In this format, you gain access to the node via the terminal/command line, and can interact with it as you would have connected via ssh directly to this compute node. Hence, you can debug your script much easier and have faster respond time. However, please keep in mind the following two disadvantages: 1. If you happen to disconnect from Lisa because of an instable connection or your computer going in stand-by mode, the job will be canceled. 2. Your job is not automatically terminated when your script has finished running, but you have to manually kill the job once you are done. If you forget it, you block the compute node for other students and waste credits that UvA payed for. Hence, only use interactive sessions for short jobs/scripts, for example, if you want to debug whether your script starts running and train the model. Do not use interactive session to train a model for a long time.

In order to start an interactive session on Lisa, you can use `srun` with the same input parameters as for the job file. For example:

```
srun --partition=gpu_shared_course --gres=gpu:1 --mem=32000M --ntasks=1 --cpus-per-
task=3 --time=00:10:00 --pty bash -i
```

This will start an interactive session with a GPU for 10 minutes. Once resources have been allocated, you will see in your terminal that you are now on one of the compute nodes, for example, `r32n5`. As a first step, you need to load the modules (`module load 2021; module load Anaconda3/2021.05`) and activate your conda environment. Then, you are ready to run your script.

4.1.4 Troubleshooting

It can happen that you encounter some issues when interacting with Lisa. A short FAQ is provided on the [SURFSara website](#), and here we provide a list of common questions/situations we have experienced from past students.

Lisa is refusing connection

It can occasionally happen that Lisa refuses the connection when you try to ssh into it. If this happens, you can first try to login to different login nodes. Specifically, try the following three login nodes:

```
ssh -X lcur____@login3.lisa.surfsara.nl  
ssh -X lcur____@login4.lisa.surfsara.nl  
ssh -X lcur____@login-gpu.lisa.surfsara.nl
```

If none of those work, you can try to use the [Pulse Secure UvA VPN](#) before connecting to Lisa. If this still does not work, then the connection issue is likely not on your side. The problem often resolves after 2-3 hours, and Lisa lets you login after it again. If the problem doesn't resolve after couple of hours, please contact your TA, and eventually the SURFSara helpdesk.

Slurm output file missing

If a job of yours is running, but no slurm output file is created, check whether the path to the output file specified in your job file actually exists. If the specified file points to a non-existing directory, no output file will be created. Note that this is not an issue by default, but you are running your job “blind” without seeing the stdout or stderr channels.

Slurm output file is empty for a long time

The slurm output file can lag behind in showing the outputs of your running job. If your job is running for couple of minutes and you would have expected a few print statements to have happened, try to flush your stdout stream ([how to flush the output in python](#)).

All my jobs are pending

With your student account, the SLURM scheduler restricts you to run only two jobs in parallel at a time. However, you can still queue more jobs that will run in sequence. This is done because with more than 200 students, Lisa could get crowded very fast if we don't guarantee a fair share of resources. If all of your jobs are pending, you can check the reason for pending in the last column of `squeue`. All reasons are listed in the `squeue` [documentation](#) under *JOB REASON CODES*. The following ones are common:

- **Priority:** There are other jobs on Lisa with a higher priority that are also waiting to be run. This means you just have to be patient.
- **QOSResourceLimit:** The job is requesting more resources than allowed. Check your job file as you are only allowed to have at max. 2 GPUs, 6 CPU cores and 125GB RAM.
- **Resources:** All nodes on Lisa are currently busy, yours will be scheduled soon.

You can also see the estimated start time of a job by running `scontrol show job JOBID`. However, note that this is the “worst case” scenario for the current number of submitted jobs, as in if all currently running jobs would need their maximum runtime. At the same time, if more people would submit their job with higher priority, yours can fall back in the queue and get a later start time.

PyTorch or other packages cannot be imported

If you run a job and see the python error message in the slurm output file that a package is missing although you have installed it in the environment, there are two things to check. Firstly, make sure to not have the environment activated on the login node when submitting the job. This can lead to an error in the anaconda module such that packages are not found on the compute node. Secondly, check that you activate the environment correctly. To verify that the correct python version is used, you can add the command `which python` before your training file. This prints out the path of the python that will be used, in which you should see the anaconda version in the dl2022 environment.

My job runs very slow

If your job executes your script much slower than you expect, check for two things: (1) have you requested a GPU and are you using it, and (2) are you using the scratch for your dataset? Not using the scratch for your dataset can create a significant communication bottleneck, especially if multiple students do it at the same time. Make sure to download or copy your dataset to the scratch and load it from there.

I am not able to use Lisa at all

If during the course, there will be major issues with Lisa (e.g. cluster goes into maintenance for a long time, issues with the filesystem, etc.), you can make use of GoogleColab, as we do already for all notebook tutorials here. All assignments do not necessarily require large amount of compute, and can often be comfortably trained on a GPU provided by GoogleColab. For an introduction to GoogleColab, see this [tutorial](#).

4.1.5 Advanced topics

Password-less login

Typing your password every time you connect to Lisa can become annoying. To enable a safe, password-less connection, you can add your public ssh key to the [SURFsara user portal](#). Next time you login from your machine to Lisa, it will only check the ssh key and not ask you for the password anymore.

Remote development with VSCode or PyCharm

The common workflow with clusters is that you first code locally and test your implementation on short runs on the CPU or evt. a local GPU, then sync your code to the cluster (e.g. via git), and finally run the full training/process on a compute node of the cluster. If you prefer to directly code on Lisa, you can do so via remote connections in tools like VSCode or PyCharm. Essentially, these IDEs can connect via SSH to Lisa, so that it looks to you like you code locally, but all code you are editing is directly saved on Lisa. Note though that to run your code, you still need to create a job script and submit it via SLURM. The login nodes, on which these IDEs connect you to, are not meant for debugging or running code. Any process that takes longer than 15 minutes will be killed (this can sometimes also include the SSH connection of VSCode/PyCharm). For more details on remote development and how to set it up, see the SURFsara documentation on [VSCode](#) and [PyCharm](#).

Tracking GPU stats

If you are curious whether you use the GPU to its full capacity, you can monitor its utilization as follows. First, you submit your job and check its job ID via `queue -u [userid]` (with your user-ID/name) or the ID that has been printed out after submitting the job via `sbatch`. Next, you can log into the node via `slurm_jobmonitor [jobid]` where you need your job ID. This gives you an interactive view on the node. Finally, you can run `nvtop` to track the GPU utilization. More details can be found [here](#).

Job Arrays

You might come into the situation where you need to run a hyperparameter search over multiple values, and don't want to write an endless number of job scripts. A much more elegant solution is a job array. Job arrays are created with two files: a job file, and a hyperparameter file. The job file will start multiple sub-jobs that each use a different set of hyperparameters, as specified in the hyperparameter file. In the job file, you need to add the argument `#SBATCH --array=....`. The argument specifies how many sub-jobs you want to start, how many to run in parallel (at maximum), and which lines to use from the hyperparameter file. For example, if we specify `#SBATCH --array=1-16%8`, this means that we start 16 jobs using the lines 1 to 16 in the hyperparameter file, and running at maximum 8 jobs in parallel at the same time. Note that the number of parallel jobs is there to limit yourself from blocking the whole cluster. However, with your student accounts, you will not be able to run more than 1 job in parallel anyways. The template job file `array_job.job` looks slightly different than the one we had before. The slurm output file is specified using `%A` and `%a`. `%A` is being automatically replaced with the job ID, while `%a` is the index of the job within the array (so 1 to 16 in our example above). Below, we also added a block for creating a checkpoint folder for the job array, and copying the job file including hyperparameters to that folder. This is good practice for ensuring reproducibility. Finally, in the training call, we specify the path checkpoint path (make sure to have implemented this argument in your argparse) with the addition of `experiment_${SLURM_ARRAY_TASK_ID}` which is a sub-folder in the checkpoint directory with the sub-job ID (1 to 16 in the example). The next line, `$(head -$SLURM_ARRAY_TASK_ID $PARAMS_FILE | tail -1)`, copies the N-th line of the hyperparameter file to this job file, and hence submits the hyperparameter arguments to the training file.

File `array_job.job`:

```
#!/bin/bash

#SBATCH --partition=gpu_shared_course
#SBATCH --gres=gpu:1
#SBATCH --job-name=ExampleArrayJob
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=3
#SBATCH --time=04:00:00
#SBATCH --mem=32000M
#SBATCH --array=1-16%8
#SBATCH --output=slurm_array_testing_%A_%a.out

module purge
module load 2021
module load Anaconda3/2021.05

# Your job starts in the directory where you call sbatch
cd $HOME/...
# Activate your environment
source activate ...

# Good practice: define your directory where to save the models, and copy the job file to it
```

(continues on next page)

(continued from previous page)

```

JOB_FILE=$HOME/.../array_job.job
HPARAMS_FILE=$HOME/.../array_job_hyperparameters.txt
CHECKPOINTDIR=$HOME/.../checkpoints/array_job_${SLURM_ARRAY_JOB_ID}

mkdir $CHECKPOINTDIR
rsync $HPARAMS_FILE $CHECKPOINTDIR/
rsync $JOB_FILE $CHECKPOINTDIR/

# Run your code
srun python -u train.py \
    --checkpoint_path $CHECKPOINTDIR/experiment_${SLURM_ARRAY_TASK_ID} \
    $(head -$SLURM_ARRAY_TASK_ID $HPARAMS_FILE | tail -1)

```

The hyperparameter file is nothing else than a text file in which each line denotes one set of hyperparameters for which you want to run an experiment. There is no specific order in which you need to put the lines, and you can extend the lines with as many hyperparameter arguments as you want.

File `array_job_hyperparameters.txt`:

```
--seed 42 --learning_rate 1e-3
--seed 43 --learning_rate 1e-3
--seed 44 --learning_rate 1e-3
--seed 45 --learning_rate 1e-3
--seed 42 --learning_rate 2e-3
--seed 43 --learning_rate 2e-3
--seed 44 --learning_rate 2e-3
--seed 45 --learning_rate 2e-3
--seed 42 --learning_rate 4e-3
--seed 43 --learning_rate 4e-3
--seed 44 --learning_rate 4e-3
--seed 45 --learning_rate 4e-3
--seed 42 --learning_rate 1e-2
--seed 43 --learning_rate 1e-2
--seed 44 --learning_rate 1e-2
--seed 45 --learning_rate 1e-2
```

Additional partitions

During the Master's program at the University of Amsterdam, you will likely gain accounts for Lisa during other courses as well, as well as during your Master thesis (if performed at university). In this case, you can get access to partitions besides `gpu_shared_course` allowing you to run more computationally heavy experiments. A full list of partitions can be found [here](#). Besides the nodes with the GeForce 1080Ti GPUs, the most commonly used partition for DL experiments is called `gpu_titanrtx` which has nodes with the following config:

Processor	CPU Cores	RAM Memory	GPUs
Gold 5118 (2.3GHz)	24	192GB	4x Titan RTX, 24 GB GDDR6

The Titan RTX are faster and provide more GPU memory than the 1080Ti's. However, these nodes cost more than twice as many credits (91.2 vs 42.1 for full node per hour, see [accounting](#)), so use them only if needed. Furthermore, in many cases, you do not need a full node and might only need a single GPU. Similar to the course partition, there exist `gpu_titanrtx_shared` and `gpu_shared` which allow you access to partial node uses, like 1 or 2 GPUs.

If you are in the need of pure CPU-based jobs, you can use the **shared** and **normal** partitions.

Additional links

Many more details on Lisa, SLURM, etc. can be found on the SURFSara wiki, as well as a different perspective on the aspects we have discussed in this tutorial. A (non-exclusive) list of useful links:

- [A crash course on Linux commands that you can run on Lisa](#)
- [How to use SSH on Windows/Mac/Linux, authenticate with SSH keys instead of passwords, and transfer files with SCP or FTP](#)
- [How to write your own job script](#)
- [How to submit/cancel a job](#)
- [How to see a job's status in the queue](#)
- [How to do remote development with VSCode and PyCharm](#)
- [Lisa partitions, hardware, and file system](#)
- [Lisa file system sharing](#)
- [Lisa usage and accounting](#)

4.2 Guide 2: Research projects with PyTorch

- Based on some feedback I got, we will try to summarize tips and tricks on how to setup and structure large research projects in PyTorch, such as your Master Thesis
- Feel free to contribute yourself if you have good ideas

4.2.1 Setup

Framework

- Choosing the right framework can be essential. If you have standard optimization loops of a single forward pass and return a loss, consider going with PyTorch Lightning. It reduces the code overhead a lot and allows to easily scale your model to multiple GPUs and/or nodes if needed. Nonetheless, if you expect that you need to change the default training procedure quite a bit, consider going with plain PyTorch and write your own framework. It might take more time initially, but makes edits in the optimization procedure easier.
- For an own framework, the following can be used as an example setup:

```
general/
    train.py
    task.py
    mutils.py
layers/
experiments/
    task1/
        train.py
        task.py
        eval.py
```

(continues on next page)

(continued from previous page)

```

dataset.py
task2/
    train.py
    task.py
    eval.py
    dataset.py

```

- The `general/train.py` file summarizes the default operations every model needs (training loop, loading/saving model, setting up model, etc.). If you use PyTorch Lightning, this reduces to a train file per task, and only needs the specification of the trainer object.
- The `general/task.py` file summarizes a template for the specific parts you have to do for a task (training step, validation step, etc.). If you use PyTorch Lightning, this would be the definition of the Lightning Module.
- The `layers/models` folder contains the code for specifying the `nn.Modules` you use for setting up the model.
- The `experiments` folder contains the task-specific code. Each task has its own `train.py` for specifying the argument parser, setting up the model, etc., while the `task.py` overwrites the template in `general/task.py`. The `eval.py` file should have as input a checkpoint directory of a trained model, and should evaluate this model on the test dataset. Finally, the file `dataset.py` contains all parts you need for setting up the dataset.
- Note that this template assumes that you might have multiple different tasks and multiple different models. If you have a simpler setup, you can inherently shrink the template together.

Argument parser

- It is a good practice to use argument parsers for specifying hyperparameters. Argument parsers allow you to call a training like `python train.py --learning ... --seed ... --hidden_size ...` etc.
- If you have multiple models to choose from, you will have multiple set of hyperparameters. A good summary on that can be found in the [PyTorch Lightning documentation](#) without the need of using Lightning. In essence, you can define a static method for each model that returns a parser for its specific hyperparameters. This makes your code cleaner and easier to define new tasks without copying the whole argument parser.
- To ensure reproducibility (more details below), it is recommended to save the arguments as a json file or similar in your checkpoint folder.

4.2.2 Hyperparameter search

- In general, hyperparameter search is all about experience. Once you have trained a lot of models, it will become easier for you to pick reasonable first-guess hyperparameters.
- The first approach to take is to look at related work to your model, and see what others have used as hyperparameters for similar models. This will help you to get started with a reasonable choice.
- Hyperparameter search can be expensive. Thus, try to do the search on shallow models first before scaling them up.
- Although a large grid search is the best way to get the optimum out of your model, it is often not reasonable to run. Try to group hyperparameters, and optimize each group one by one.

Toolkits

- PyTorch Lightning provides a lot of useful tricks and toolkits on hyperparameter searching, such as:
 - Learning rate [finder](#) that plots the learning rate vs loss for a few initial batches, and helps you to choose a reasonable learning rate.
 - Autoscaling batch sizes which finds the largest possible batch size given your GPU (helpful if you have very deep, large models, and it is obvious you need the largest batch size possible).
- For comparing multiple hyperparameter configurations, you can add them to TensorBoard. This is a clean way of comparing multiple runs. If interested, a blog on this can be found [here](#).
- There are multiple libraries that support you in automatic hyperparameter search. A good overview for those in PyTorch can be found [here](#).

Reproducibility

- Everything is about reproducibility. Make sure you can reproduce any training you do with the same random values, batches, etc. You will come to a point where you have tried a lot of different approaches, but none were able to improve upon one of your previous runs. When you try to run the model again with the best hyperparameters, you don't want to have a bad surprise (believe me, enough people have this issue, and it might also happen to you). Hence, before starting any grid search, make sure you are able to reproduce runs. Run two jobs in parallel on Lisa with the same hyperparams, seeds, etc., and if you don't get the exact same results, stop and try to fix it before anything else.
- Another fact about reproducibility is that saving and loading a model works without any problems. Make sure before a long training that you are able to load a saved model from the disk, and achieve the exact same test score as you had during training.
- Print your hyperparameters into the SLURM output file (simple print statement in python). This will help you identifying the runs, and you can easily check whether Lisa executes the job you intended to. Further, hyperparameters should be stored in a separate file in your checkpoint directory, whether saved by PyTorch Lightning or yourself.
- When running a job, copy the job file automatically to your checkpoint folder. This improves reproducibility by ensuring you have the exact running command ready.
- Besides the slurm output file, create a output file in which you store the best training, validation and test score. This helps you when you want to quickly compare multiple models or create statistics of your results.
- If you want to be on the safe side and use git, you can even print/save the hash of the git commit you are currently on, and any changes you had made to the files. An example of how to do this can be found [here](#).

Seeds

- DL models are inherently noisy, and no two runs are the same if you don't ensure a deterministic execution. Before running a grid search, try to get a feeling of how noisy your experiments might be. The more noise you expect compared to your result scale, the more versions of your model you need to run to get a statistically significant difference between settings.
- After finishing the grid search, run another model of the best configuration with a new seed. If the score is still the best, take the model. If not, consider running a few more seeds for the top k models in your grid search. Otherwise, you risk taking a suboptimal model, which was just lucky to be the best for a specific seed.

Learning rate

- The learning rate is an important parameter, which depends on the optimizer, the model, and many more other hyperparameters.
- A usual good starting point is 0.1 for SGD, and 1e-3 for Adam.
- The deeper the model is, the lower the learning rate usually should be. For instance, Transformer models usually apply learning rates of 1e-5 to 1e-4 for Adam.
- The lower your batch, the lower the learning rate should be. Consider using [gradient accumulation](#) if your batch size is getting too small (PyTorch Lightning supports this, see [here](#)).
- Consider using the PyTorch Lightning [learning rate finder](#) toolkit for an initial good guess.

LR scheduler

- Similarly to the learning rate, the scheduler to apply again depends on the classifier and model.
- For image classifiers and SGD as optimizer, the multi-step LR scheduler has shown to be good choice.
- Models trained with Adam commonly use a smooth exponential decay in the learning rate or a cosine-like scheduler.
- For Transformers: remember to use a learning rate warmup. The cosine scheduler is often used for decaying the learning rate afterwards, but can also be replaced by an exponential decay.

Regularization

- Regularization is important in networks if you see a significantly higher training performance than test performance.
- The regularization parameters all interact with each other, and hence must be tuned together. The most commonly used regularization techniques are:
 - Weight decay
 - Dropout
 - Augmentation
- Dropout is usually a good idea as it is applicable to most architectures and has shown to effectively reduce overfitting.
- If you want to use weight decay in Adam, remember to use `torch.optim.AdamW` instead of `torch.optim.Adam`.

Domain specific regularization

- There are couple of regularization techniques that depend on your input data/domain. The most common include:
 - Computer Vision: image augmentation like horizontal flip, rotation, scale-and-crop, color distortion, gaussian noise, etc.
 - NLP: input dropout of whole words.
 - Graphs: dropping edges, nodes, or part of the features of all nodes.

Grid search with SLURM

- SLURM supports you to do a grid search with [job arrays](#). We have discussed job arrays in the [Lisa guide](#).
- Job arrays allow you to start N jobs in parallel, each running with slightly different settings.
- It is effectively the same as creating N job files and calling N times `sbatch ...`, but this can become annoying and is messy at some point.

PyTorch Lightning

Writing the job arrays can be sometimes annoying, and hence it is advised to write a script that can automatically generate the hyperparameter files if you have to do this often enough (for instance, by adding the seed parameter 4 times to each other hyperparam config). However, if you are using PyTorch Lightning, you can directly create a job array file. The documentation for this can be found [here](#).

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on [GitHub](#).

4.3 Guide 3: Debugging in PyTorch

When you start learning PyTorch, it is expected that you hit bugs and errors. To help you debug your code, we will summarize the most common mistakes in this guide, explain why they happen, and how you can solve them.

4.3.1 My model is underperforming

Situation: Your model is not reaching the performance that it should, but PyTorch is not telling you why that happens. These errors are the most annoying bugs since those can be hard to debug. Nonetheless, there are couple of things you can check. If none of these solve the problem for you, one of us TAs will help you debug your code more in detail.

Softmax, CrossEntropyLoss and NLLLoss

The most common mistake is the mismatch between loss function and output activation function. The loss module `nn.CrossEntropyLoss` in PyTorch performs two operations: `nn.LogSoftmax` and `nn.NLLLoss`. Hence, the input to this loss module should be the output of your last linear layer. **Do not apply a softmax before the Cross-Entropy loss.** Otherwise, PyTorch will apply a log-softmax on your softmax outputs, which will significantly worsen the performance, and give you headaches.

If you use the loss module `nn.NLLLoss`, you need to apply the log-softmax yourself. **NLLLoss requires log-probabilities, not plain probabilities.** Hence, make sure to apply `nn.LogSoftmax` or `nn.functional.log_softmax`, and **not** `nn.Softmax`.

Softmax over the right dimension

Pay attention to the dimension you apply your softmax over. Usually, this is the last dimension of your output tensor, which you can identify with e.g. `nn.Softmax(dim=-1)`. If you mix up the dimension, your model ends up with random predictions.

Categorical data and Embedding

Categorical data, as for example language characters or the datasets you are given in assignment 2, require special care. Data like language characters ‘a’, ‘b’, ‘c’ etc. are usually represented as integers 0, 1, 2, etc. **Do not use integers as input for categorical data.** If you would enter those integers as inputs to the model, two problems arise.

1. You bias the model to see relations where there are none. In the language example above, the model would think that ‘a’ is closer to ‘b’ than to ‘o’, although ‘a’ and ‘o’ are both vocals, and the closeness of ‘a’ and ‘b’ does not necessarily say anything about their usage.
2. If you have many categories, you will have input values between 0 and >50. The model will have a hard time separating all those >50 categories without blending over some. Hence, the model loses a lot of information although this is not necessary.

The much better option in the case of categorical data is to use one-hot vectors, or embeddings. A one-hot vector represents each category by a vector of 0s, with one index being 1. This makes the model’s life much easier as it can distinguish between the categories in a very simple manner (if feature !=0, it is a specific category). Alternatively, you can learn an embedding with the help of `nn.Embedding`. The inputs to this module are:

- `num_embeddings` which is the number of different categories you have in your input data (in case of language characters, something like 26 as you have ‘a’ to ‘z’)
- `embedding_dim` which is the number of features you want to represent each category with. If you use the embedding directly as input to an LSTM or RNN, a good rule of thumb is to use 1/4 - 1/2 of your hidden size inside the LSTM.
- `padding_idx` which would allow you to assign a specific index for the padding symbol. Can be skipped if you do not use “-1” as padding index.

The embedding feature vectors are randomly initialized from $\mathcal{N}(0, 1)$. **Do not overwrite this init by Kaiming, Xavier or similar.** The used standard deviation is 1 because the initialization, activation functions etc. have been designed to have a input standard deviation of 1. Example usage of the embedding module:

```
[1]: import torch
import torch.nn as nn
# Create 5 embedding vectors each with 32 features
embedding = nn.Embedding(num_embeddings=5,
                        embedding_dim=32)

# Example integer input
input_tensor = torch.LongTensor([[0, 4], [2, 3], [0, 1]])

# Get embeddings
embed_vectors = embedding(input_tensor)

print("Input shape:", input_tensor.shape)
print("Output shape:", embed_vectors.shape)
print("Example features:\n", embed_vectors[:, :, :2])

Input shape: torch.Size([3, 2])
Output shape: torch.Size([3, 2, 32])
```

(continues on next page)

(continued from previous page)

Example features:

```
tensor([[[ 0.0504, -0.2422],
       [ 0.0342,  0.2217]],

      [[ 2.7813, -0.3641],
       [-0.0981,  0.4069]],

      [[ 0.0504, -0.2422],
       [ 1.2092,  0.1760]]], grad_fn=<SliceBackward>)
```

The `nn.Embedding` object is a module like a linear layer or convolution. Thus, it needs to be defined in the `__init__` function of your higher-level module. **Do not create the Embedding module in the forward pass.** Otherwise, you will have different embeddings every time you run the model, and hence, your model is not able to learn.

Time dimension in nn.LSTM

By default, PyTorch's `nn.LSTM` module assumes the input to be sorted as `[seq_len, batch_size, input_size]`. Make sure that you do not confuse the sequence length and batch dimension. The LSTM would still run without an error, but will give you wrong results. If you want to change this behavior to accepting an input shape of `[batch_size, seq_len, input_size]`, you can specify the argument `batch_first=True` when creating the LSTM object. Have a closer look at the [documentation](#) for details.

Hidden shape mismatch

If you perform matrix multiplications and have a shape mismatch between two matrices, PyTorch will complain and throw an error. However, there are also situations where PyTorch does not throw an error because the misaligned dimensions have (unluckily) the same size. For instance, imagine you have a weight matrix of size $d_{in} \times d_{out}$. If you take the input x of size $B \times d_{in}$ (B being the batch dimension), and in your hyperparameter setting, $B = d_{in}$, you can end up performing the matrix multiplication over the wrong dimension while PyTorch is not detecting it. **Test your code with multiple, different batch sizes to prevent shape misalignments with the batch dimension.**

Training and Evaluation switch

In PyTorch, a module and/or neural network has two modes: training and evaluation. You switch between them using `model.eval()` and `model.train()`. The modes decide for instance whether to apply dropout or not, and how to handle the forward of Batch Normalization. However, a common mistake is to forget to set your model back into training mode after evaluation. **Make sure to set your model back to train mode after validation.** In case your model does not contain dropout, BatchNorm or similar modules, this might not affect your performance.

Parameter handling

As you might know from the PyTorch Tutorial, PyTorch supports hierarchical usage of `nn.Modules`. One module can contain another module, which can again contain a module, and so on. When you call `.parameters()` on a module, PyTorch looks for all modules inside the module to also add their parameters to the highest-level module's parameter. However, **PyTorch does not detect parameters of modules in lists, dicts or similar structures.** If you have a list of modules, make sure to put them into a `nn.ModuleList` or `nn.Sequential` object. Parameters of modules inside those containers are detected. Similarly, for dictionaries, you can use `nn.ModuleDict`.

Parameters and “.to(device)”

To push your model and/or data to GPU, you can use `.to(device)` where `device` is an device object or string (“cpu” for CPU-only machines, and “cuda”/“cuda:0” for GPUs). However, **do not call ``.to(device)`` during parameter init.** If you define a parameter like `self.W = nn.Parameter(torch.Tensor(64, 128)).to(device)`, your model will not register the parameter on GPU because the “`.to`” operator creates a new Tensor. Parameters, nonetheless, have to be leaf Tensors, hence your parameters will not be recognized (corresponding [GitHub issue](#)). It is much better practice to only call `.to(device)` once *after* finishing the init of the model, and not inside the model.

My model runs fine on CPU, but gets NaN loss on GPU

If this is the case, you likely have the bug of parameters and `.to(device)` as explained above.

Initialization

Initializing the parameters of your model correctly is very important (see [Tutorial 4](#) for details on this). Initializing parameters with a standard normal distribution is not a good practice and often fails. It can occassionally work for very shallow networks, but don’t risk it! **Think about your initialization, and use proper methods like Kaiming or Xavier.**

Zero-grad in optimizers

Remember to call `optimizer.zero_grad()` before doing `loss.backward()`. If you do not reset the gradients for all parameters before performing backpropagation, your gradients will be added to those from the previous batch. Hence, your gradients end up to be not the ones you intended them to be.

Weight decay and Adam

Adam is known to have a different implementation of weight decay in many frameworks than you would expect. Specifically, the weight decay is usually added as gradients **before** determining the adaptive learning rate, and hence scaling up the weight decay for parameters with low gradient norms. Details on this problem, which is actually shared across most common DL frameworks, can be found [here](#). In PyTorch, you can use the desired version of weight decay in Adam using `torch.optim.AdamW` (identical to `torch.optim.Adam` besides the weight decay implementation).

Check your metric calculation

This might sound a bit stupid but check your metric calculation twice or more often before doubting yourself or your model. Metrics like accuracy are easy to calculate, but it is as easy to add a bug into the code. For instance, check that you are averaging over the batch dimension and not accidentally over the class dimension or any other.

My bits per dimension score is very low

If you obtain a very low bits per dimension score for likelihood-based generative models after already the first iteration, the calculation might not be fully correct. Specifically, the negative log likelihood input to the bpd-metric function is expected to be the **sum** of the individual pixel’s log likelihood of an image, not the mean. The mean is taken inside the `bpd` function.

4.3.2 PyTorch throws an error

These errors are the easier bugs to correct because PyTorch actually talks to you about what is wrong. Until these are not solved, you probably cannot train your model.

Trying to backward through the graph a second time, specify retain_graph=True

This error occurs if you re-use a tensor from the computation graph of the previous batch. This should usually not happen. Make sure to not keep tensors across batches if not strictly necessary. Example where this issue can occur: when implementing your own LSTM, make sure that the initial hidden state is a constant zero tensor, and not the last hidden state of the previous batch.

Size mismatch

This usually occurs if your dimension of the input to a module does not match the specified input dimension of the weight tensor, like in a linear layer. Make sure to have specified the correct dimensions. Usually, a good way to debug this is to print the shape of the input tensor before every layer you call.

If this happens for a matrix multiplication you have implemented, print the shapes of both matrices, and try to figure out over which dimension the matrix multiplication should actually have been performed, and over which PyTorch currently does it.

Device mismatch

You might sometimes see a mistake such as: `Runtime Error: Input type (torch.FloatTensor) and weight type (torch.cuda.FloatTensor) should be on the same device`. This error indicates that the input data is on CPU, while your weights are on the GPU. Make sure that all data is on the same device. This is usually the GPU as it supports acceleration for both training and testing.

4.3.3 Good practices

There are many good practices in PyTorch. We try to add a few below that might make your life easier. Another list of good practices can be found [here](#).

Use nn.Sequential and nn.ModuleList

If you have a model with a lot of layers, you might want to summarize them into a `nn.Sequential` or `nn.ModuleList` object. In the forward pass, you only need to call the sequential, or iterate through the module list. A MLP can be implemented as follows:

```
[2]: class MLP(nn.Module):
    def __init__(self, input_dims=64, hidden_dims=[128, 256], output_dims=10):
        super().__init__()
        hidden_dims = [input_dims] + hidden_dims
        layers = []
        for idx in range(len(hidden_dims)-1):
            layers += [
```

(continues on next page)

(continued from previous page)

```

        nn.Linear(hidden_dims[i], hidden_dims[i+1]),
        nn.ReLU(inplace=True)
    ]
self.layers = nn.Sequential(*layers)

def forward(self, x):
    return self.layers(x)

```

In-place activation functions

Some activation functions like `nn.ReLU` or `nn.LeakyReLU` have the argument `inplace`. By default it is `False`, but it is recommended to set it to `True` in neural networks. What it does is that the forward pass overwrites the original values of the input with the new output. This option is only available for activation functions where we do not need to know the original input for backpropagation. For instance, in ReLU, the values that are set to zero, have a gradient of zero independent of its specific input value. In-place operation can save a bit of memory, especially if you have large feature maps.

Create modules for repeating blocks

In deep neural networks, you usually have blocks that are repeatedly added to the model. If those blocks require a more complex forward function than just `x = layer(x)`, it is recommended to implement them in a separate module. For example, a ResNet consists of multiple ResNet blocks with a residual connection. The ResNet blocks apply a small neural network, and add the output back to the input. It is better to implement this dynamic in a separate `nn.Module` class to keep the main model class small and clear.

Stack layers/weights with same input

If you have multiple linear layers or convolutions that have the same input, you can stack them together to increase efficiency. Suppose we have two layers on x : $y_1 = W_1x + b_1$, $y_2 = W_2x + b_2$. While you could implement it by two linear layers, you can get the exact same neural network by stacking the two layers into one. The single layer is more efficient as this represents a single matrix operation instead of two for the GPU, and hence we can parallelize the computation. An example is shown below:

```
[3]: x = torch.randn(2, 10)

# Implementation of separate layers:
y1_layer = nn.Linear(10, 20)
y2_layer = nn.Linear(10, 30)
y1 = y1_layer(x)
y2 = y2_layer(x)

# Implementation of a stacked layer:
y_layer = nn.Linear(10, 50)
y = y_layer(x)
y1, y2 = y[:, :20], y[:, 20:50]
```

If you implement the linear layer manually, you need to stack the weight and bias tensor accordingly. Note that you should change your initialization for the stacked case if necessary. If your initialization depends on the output size of

a layer (as for example in Xavier), you would get a different standard deviation for initialization in the two implementations. Still, if your initialization solely depends on the input dimension (e.g. Kaiming), no change is necessary. An example case where this stacking can be beneficial is LSTMs as all four gates use the exact same input.

Use loss functions on logits

Classification loss functions such as Binary Cross Entropy have two versions in PyTorch: with and without logits. It is recommended and good practice to use the loss functions on logits. This is because it is numerically more stable and prevents any instabilities when your model is very wrong in its prediction. If you do not use the logit loss functions, you might run into problems when the model predicts very high or low values that are not correct. In BCE, you will then encounter a log over a value very close to 0. If you are lucky, you just get a very high number (and your model might still diverge because of this), or actually end up with NaN values.

Make use of `torch.nn.functional`

You do not always need modules. Many methods that do not have parameters are implemented as both modules and functions (e.g. log-softmax/softmax, binary cross entropy, etc.). If you need a softmax but do not have a `nn.Sequential` where you could add it to, the function option `F.softmax(..., dim=...)` is cleaner than defining a separate module first.

Clip gradient norms

Another good training practice is to clip gradient norms. Even if you set a high threshold, it can stop your model from diverging, even when it gets very high losses. While in MLPs not strictly necessary, RNNs, Transformers, and likelihood models can often benefit from gradient norm clipping. In PyTorch, you can use it via `torch.nn.utils.clip_grad_norm_(...)` (remember to call it after `loss.backward()` but before `optimizer.step()`). In PyTorch Lightning, you can set the clipping norm via `gradient_clip_val=...` in the Trainer.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.4 Guide 4: Research Projects with JAX

Filled notebook:

Author: Phillip Lippe

This guide summarizes some tips, tricks and practices that are useful when working with JAX for a research project. In my opinion, one key aspect that JAX is missing compared to PyTorch is a framework like [PyTorch Lightning](#) that can massively reduce code overhead while still being flexible enough for supporting almost any model/task. Although there exist such libraries for certain common tasks, like [trax](#) or [scenic](#) (attention-based CV), I have not come across one so far which was sufficiently flexible for my research. Hence, in this guide, we build a simpler version of a PyTorch Lightning trainer, that summarizes all training, logging, etc. behavior that we need for almost any model, and allows training various models with much fewer lines than from scratch. Moreover, we implement some simple examples to

showcase possible training structures, and underline its flexibility by performing automatic hyperparameter tuning with Optuna. Since this guide will be about code structures, it is more code-heavy than the other guides and can also be run in Google Colab if preferred.

First, let's import some standard libraries. For this guide, we will use the data loading functionalities of PyTorch, but one could also use the TensorFlow dataset API. Additionally, we integrate loggers from PyTorch Lightning since they support a flexible API and have most popular logging application implemented (e.g. TensorBoard, Weights and Biases).

```
[1]: # Standard libraries
import os
import sys
from typing import Any, Sequence, Optional, Tuple, Iterator, Dict, Callable, Union
import json
import time
from tqdm.auto import tqdm
import numpy as np
from copy import copy
from glob import glob
from collections import defaultdict

# JAX/Flax
# If you run this code on Colab, remember to install flax and optax
# !pip install --quiet --upgrade flax optax
import jax
import jax.numpy as jnp
from jax import random
from flax import linen as nn
from flax.training import train_state, checkpoints
import optax

# PyTorch for data loading
import torch
import torch.utils.data as data

# Logging with Tensorboard or Weights and Biases
# If you run this code on Colab, remember to install pytorch_lightning
# !pip install --quiet --upgrade pytorch_lightning
from pytorch_lightning.loggers import TensorBoardLogger, WandbLogger
```

4.4.1 Trainer module for JAX with Flax

As seen in previous tutorials, Flax gives us already some basic functionalities for training models. One part of it is the TrainState, which holds the model parameters and optimizers, and allows updating it. However, there might be more model aspects that we would like to add to the TrainState. For instance, if a model uses Batch Normalization as in Tutorial 5, we need to keep the batch statistics in order to evaluate the models on a test dataset. Furthermore, many models contain stochastic elements such as dropout or sampling in generative models (e.g. Normalizing Flows). Thus, we extend the TrainState class from Flax to also include the batch statistics as `batch_stats` and a pseudo-random number generation `rng`. Note that if models do not require these elements, they can simply be `None` without breaking our code.

```
[2]: class TrainState(train_state.TrainState):
    # A simple extension of TrainState to also include batch statistics
```

(continues on next page)

(continued from previous page)

```
# If a model has no batch statistics, it is None
batch_stats : Any = None
# You can further extend the TrainState by any additional part here
# For example, rng to keep for init, dropout, etc.
rng : Any = None
```

Now we already come to the main part of this guide: the Trainer module for JAX/Flax. The shown module here is not meant to be the ‘one and only’ way of doing it, and is more meant as showcasing one possible option of obtaining a Lightning-like API in JAX. The module can easily be extended by more functionalities, depending on what is needed/preferred by the individual users.

First let’s make a list of functionalities that we would want the Trainer module to include:

- **Logging:** For basically all usecases and models, we want to log our hyperparameters, training/validation performance, and model checkpoints. For the second point, we can make use of PyTorch Lightning’s logger classes like `TensorBoardLogger` and `WandbLogger`. For the model checkpoints, we use `flax.checkpoints`. In terms of flexibility, the trainer should support arbitrary sets of hyperparameters, since different models may require different hyperparameters. Similarly, it should be easy to add new metrics for logging, like accuracy for classification or intersection over union for segmentation.
 - Implemented in: `init_logger`, `save_model`, `load_model`, `save_metrics`
- **Model state initialization:** In contrast to PyTorch, JAX separates the model itself from the learnable parameters. Creating a set of parameters for a model requires some boiler-template code, like creating a PRNG for the parameter generation and creating an initial `TrainState`. At the same time, we need to allow overwriting the `model.init` code, since different architectures will have different input arguments for the forward pass (e.g. models with dropout require a dropout-PRNG).
 - Implemented in: `init_model`, `run_model_init`, `print_tabulate`
- **Optimizer initialization:** Following with the parameter initialization, we also need to create an optimizer and its eventual parameters (e.g. momentum and adaptive learning rate parameters in Adam). Since most models use a similar set of optimizers (SGD or Adam) and extra functionalities like gradient clipping and learning rate scheduling, we can write a template method that creates an optimizers based on some hyperparameters. However, it should be possible to overwrite this method if very specific optimizer settings/learning rate schedulers are needed. Since some schedulers require information about the overall number of training iterations, we create the optimizer right before starting the training.
 - Implemented in: `init_optimizer`
- **Training loop:** Most models follow a similar training procedure where we train a model for several epoch on the training dataset, and evaluate it in between on the validation dataset. If a model is better than all previous models, we want to save its weight for loading them potentially later. Importantly, however, each model will have a very different training and validation step. Thus, similarly to PyTorch Lightning, we expect that an inheriting Trainer module has to define a training step function and evaluation step function, that can be jitted and used in the training loop. This is implemented in the function `train_model`, `train_epoch`, `eval_model`, `create_functions`, `create_jitted_functions`. Additional aspects to consider include:
 - Whether a model is better than the previous ones or not depends on the task at hand. For example, classification models are usually compared by their accuracy, trying to achieve the maximum value, while regression models aim for the lowest loss. Hence, we need a flexible API to support different ways of comparing models and finding the best one. Implemented in: `is_new_model_better`
 - Within the training loop, we might want to perform additional operations, like logging reconstruction examples of an autoencoder after every few epochs. To do so, PyTorch Lightning provides functions that are called at different stages during training, which we can similarly integrate in our Trainer module. Implemented in: `on_training_start`, `on_training_epoch_end`, `on_validation_epoch_end`

- Depending on whether we run the model on a cluster with no display or on our local machine, we might want to see progress bars that track the training progress. Hence, the Trainer module should have to switch to enable or disable these progress bars. Implemented in: `tracker`
- **Inference:** After we have finished training, we might want to load a model at a later time and perform inference experiments with it. For example, in [Tutorial 9](#), we use a trained autoencoder for an image search engine. To support this, two functionalities are needed: (1) loading a model from disk, including its hyperparameters (i.e. the function `load_from_checkpoint` in PyTorch Lightning), and (2) binding parameters to a model to reduce code overhead. Both parts can be implemented in our Trainer module.
 - Implemented in: `load_from_checkpoint`, `bind_model`

With these requirements in mind, let's finally implement the module. Note that it is a considerably long code cell since we want to support many different settings. We recommend to take some time to go through the code and understand how all the elements are implemented, and how one can extend it depending on their own needs.

[3]: `class TrainerModule:`

```
def __init__(self,
            model_class : nn.Module,
            model_hparams : Dict[str, Any],
            optimizer_hparams : Dict[str, Any],
            exmp_input : Any,
            seed : int = 42,
            logger_params : Dict[str, Any] = None,
            enable_progress_bar : bool = True,
            debug : bool = False,
            check_val_every_n_epoch : int = 1,
            **kwargs):
    """
    A basic Trainer module summarizing most common training functionalities
    like logging, model initialization, training loop, etc.
    """

    Atributes:
    model_class: The class of the model that should be trained.
    model_hparams: A dictionary of all hyperparameters of the model. Is
        used as input to the model when created.
    optimizer_hparams: A dictionary of all hyperparameters of the optimizer.
        Used during initialization of the optimizer.
    exmp_input: Input to the model for initialization and tabulate.
    seed: Seed to initialize PRNG.
    logger_params: A dictionary containing the specification of the logger.
    enable_progress_bar: If False, no progress bar is shown.
    debug: If True, no jittering is applied. Can be helpful for debugging.
    check_val_every_n_epoch: The frequency with which the model is evaluated
        on the validation set.
    """

    super().__init__()
    self.model_class = model_class
    self.model_hparams = model_hparams
    self.optimizer_hparams = optimizer_hparams
    self.enable_progress_bar = enable_progress_bar
    self.debug = debug
    self.seed = seed
    self.check_val_every_n_epoch = check_val_every_n_epoch
```

(continues on next page)

(continued from previous page)

```

self.exmp_input = exmp_input
# Set of hyperparameters to save
self.config = {
    'model_class': model_class.__name__,
    'model_hparams': model_hparams,
    'optimizer_hparams': optimizer_hparams,
    'logger_params': logger_params,
    'enable_progress_bar': self.enable_progress_bar,
    'debug': self.debug,
    'check_val_every_n_epoch': check_val_every_n_epoch,
    'seed': self.seed
}
self.config.update(kwargs)
# Create empty model. Note: no parameters yet
self.model = self.model_class(**self.model_hparams)
self.print_tabulate(exmp_input)
# Init trainer parts
self.init_logger(logger_params)
self.create_jitted_functions()
self.init_model(exmp_input)

def init_logger(self,
                logger_params : Optional[Dict] = None):
    """
    Initializes a logger and creates a logging directory.

    Args:
        logger_params: A dictionary containing the specification of the logger.
    """
    if logger_params is None:
        logger_params = dict()
    # Determine logging directory
    log_dir = logger_params.get('log_dir', None)
    if not log_dir:
        base_log_dir = logger_params.get('base_log_dir', 'checkpoints/')
        # Prepare logging
        log_dir = os.path.join(base_log_dir, self.config["model_class"])
        if 'logger_name' in logger_params:
            log_dir = os.path.join(log_dir, logger_params['logger_name'])
        version = None
    else:
        version = ''
    # Create logger object
    logger_type = logger_params.get('logger_type', 'TensorBoard').lower()
    if logger_type == 'tensorboard':
        self.logger = TensorBoardLogger(save_dir=log_dir,
                                         version=version,
                                         name=' ')
    elif logger_type == 'wandb':
        self.logger = WandbLogger(name=logger_params.get('project_name', None),
                                 save_dir=log_dir,
                                 version=version,

```

(continues on next page)

(continued from previous page)

```

        config=self.config)
else:
    assert False, f'Unknown logger type \'{logger_type}\''
# Save hyperparameters
log_dir = self.logger.log_dir
if not os.path.isfile(os.path.join(log_dir, 'hparams.json')):
    os.makedirs(os.path.join(log_dir, 'metrics/'), exist_ok=True)
    with open(os.path.join(log_dir, 'hparams.json'), 'w') as f:
        json.dump(self.config, f, indent=4)
self.log_dir = log_dir

def init_model(self,
               exmp_input : Any):
"""
Creates an initial training state with newly generated network parameters.

Args:
    exmp_input: An input to the model with which the shapes are inferred.
"""
# Prepare PRNG and input
model_rng = random.PRNGKey(self.seed)
model_rng, init_rng = random.split(model_rng)
exmp_input = [exmp_input] if not isinstance(exmp_input, (list, tuple)) else exmp_
input
# Run model initialization
variables = self.run_model_init(exmp_input, init_rng)
# Create default state. Optimizer is initialized later
self.state = TrainState(step=0,
                        apply_fn=self.model.apply,
                        params=variables['params'],
                        batch_stats=variables.get('batch_stats'),
                        rng=model_rng,
                        tx=None,
                        opt_state=None)

def run_model_init(self,
                   exmp_input : Any,
                   init_rng : Any) -> Dict:
"""
The model initialization call

Args:
    exmp_input: An input to the model with which the shapes are inferred.
    init_rng: A jax.random.PRNGKey.

Returns:
    The initialized variable dictionary.
"""
return self.model.init(init_rng, *exmp_input, train=True)

def print_tabulate(self,
                  exmp_input : Any):

```

(continues on next page)

(continued from previous page)

```

"""
Prints a summary of the Module represented as table.

Args:
    exmp_input: An input to the model with which the shapes are inferred.
"""

print(self.model.tabulate(random.PRNGKey(0), *exmp_input, train=True))

def init_optimizer(self,
                   num_epochs : int,
                   num_steps_per_epoch : int):
"""
Initializes the optimizer and learning rate scheduler.

Args:
    num_epochs: Number of epochs the model will be trained for.
    num_steps_per_epoch: Number of training steps per epoch.
"""

hparams = copy(self.optimizer_hparams)

# Initialize optimizer
optimizer_name = hparams.pop('optimizer', 'adamw')
if optimizer_name.lower() == 'adam':
    opt_class = optax.adam
elif optimizer_name.lower() == 'adamw':
    opt_class = optax.adamw
elif optimizer_name.lower() == 'sgd':
    opt_class = optax.sgd
else:
    assert False, f'Unknown optimizer "{opt_class}"'
# Initialize learning rate scheduler
# A cosine decay scheduler is used, but others are also possible
lr = hparams.pop('lr', 1e-3)
warmup = hparams.pop('warmup', 0)
lr_schedule = optax.warmup_cosine_decay_schedule(
    init_value=0.0,
    peak_value=lr,
    warmup_steps=warmup,
    decay_steps=int(num_epochs * num_steps_per_epoch),
    end_value=0.01 * lr
)
# Clip gradients at max value, and evt. apply weight decay
transf = [optax.clip_by_global_norm(hparams.pop('gradient_clip', 1.0))]
if opt_class == optax.sgd and 'weight_decay' in hparams: # wd is integrated in adamw
    transf.append(optax.add_decayed_weights(hparams.pop('weight_decay', 0.0)))
optimizer = optax.chain(
    *transf,
    opt_class(lr_schedule, **hparams)
)
# Initialize training state
self.state = TrainState.create(apply_fn=self.state.apply_fn,

```

(continues on next page)

(continued from previous page)

```

        params=self.state.params,
        batch_stats=self.state.batch_stats,
        tx=optimizer,
        rng=self.state.rng)

def create_jitted_functions(self):
    """
    Creates jitted versions of the training and evaluation functions.
    If self.debug is True, not jitting is applied.
    """
    train_step, eval_step = self.create_functions()
    if self.debug: # Skip jitting
        print('Skipping jitting due to debug=True')
        self.train_step = train_step
        self.eval_step = eval_step
    else:
        self.train_step = jax.jit(train_step)
        self.eval_step = jax.jit(eval_step)

    def create_functions(self) -> Tuple[Callable[[TrainState, Any], Tuple[TrainState, Dict]], Callable[[TrainState, Any], Tuple[TrainState, Dict]]]:
        """
        Creates and returns functions for the training and evaluation step. The
        functions take as input the training state and a batch from the train/
        val/test loader. Both functions are expected to return a dictionary of
        logging metrics, and the training function a new train state. This
        function needs to be overwritten by a subclass. The train_step and
        eval_step functions here are examples for the signature of the functions.
        """
        def train_step(state : TrainState,
                      batch : Any):
            metrics = {}
            return state, metrics
        def eval_step(state : TrainState,
                      batch : Any):
            metrics = {}
            return metrics
        raise NotImplementedError

    def train_model(self,
                    train_loader : Iterator,
                    val_loader : Iterator,
                    test_loader : Optional[Iterator] = None,
                    num_epochs : int = 500) -> Dict[str, Any]:
        """
        Starts a training loop for the given number of epochs.

        Args:
            train_loader: Data loader of the training set.
            val_loader: Data loader of the validation set.
        """

```

(continues on next page)

(continued from previous page)

`test_loader: If given, best model will be evaluated on the test set.`

`num_epochs: Number of epochs for which to train the model.`

Returns:

A dictionary of the train, validation and evt. test metrics for the best model on the validation set.

"""

```
# Create optimizer and the scheduler for the given number of epochs
self.init_optimizer(num_epochs, len(train_loader))
# Prepare training loop
self.on_training_start()
best_eval_metrics = None
for epoch_idx in self.tracker(range(1, num_epochs+1), desc='Epochs'):
    train_metrics = self.train_epoch(train_loader)
    self.logger.log_metrics(train_metrics, step=epoch_idx)
    self.on_training_epoch_end(epoch_idx)
    # Validation every N epochs
    if epoch_idx % self.check_val_every_n_epoch == 0:
        eval_metrics = self.eval_model(val_loader, log_prefix='val/')
        self.on_validation_epoch_end(epoch_idx, eval_metrics, val_loader)
        self.logger.log_metrics(eval_metrics, step=epoch_idx)
        self.save_metrics(f'eval_epoch_{str(epoch_idx).zfill(3)}', eval_metrics)
    # Save best model
    if self.is_new_model_better(eval_metrics, best_eval_metrics):
        best_eval_metrics = eval_metrics
        best_eval_metrics.update(train_metrics)
        self.save_model(step=epoch_idx)
        self.save_metrics('best_eval', eval_metrics)
    # Test best model if possible
if test_loader is not None:
    self.load_model()
    test_metrics = self.eval_model(test_loader, log_prefix='test/')
    self.logger.log_metrics(test_metrics, step=epoch_idx)
    self.save_metrics('test', test_metrics)
    best_eval_metrics.update(test_metrics)
# Close logger
self.logger.finalize('success')
return best_eval_metrics
```

def train_epoch(self,
 train_loader : Iterator) -> Dict[str, Any]:

"""

Trains a model for one epoch.

Args:

`train_loader: Data loader of the training set.`

Returns:

A dictionary of the average training metrics over all batches for logging.

"""

```
# Train model for one epoch, and log avg loss and accuracy
```

(continues on next page)

(continued from previous page)

```

metrics = defaultdict(float)
num_train_steps = len(train_loader)
start_time = time.time()
for batch in self.tracker(train_loader, desc='Training', leave=False):
    self.state, step_metrics = self.train_step(self.state, batch)
    for key in step_metrics:
        metrics['train/' + key] += step_metrics[key] / num_train_steps
metrics = {key: metrics[key].item() for key in metrics}
metrics['epoch_time'] = time.time() - start_time
return metrics

def eval_model(self,
               data_loader: Iterator,
               log_prefix: Optional[str] = '') -> Dict[str, Any]:
"""
Evaluates the model on a dataset.

Args:
    data_loader: Data loader of the dataset to evaluate on.
    log_prefix: Prefix to add to all metrics (e.g. 'val/' or 'test/')

Returns:
    A dictionary of the evaluation metrics, averaged over data points
    in the dataset.
"""
# Test model on all images of a data loader and return avg loss
metrics = defaultdict(float)
num_elements = 0
for batch in data_loader:
    step_metrics = self.eval_step(self.state, batch)
    batch_size = batch[0].shape[0] if isinstance(batch, (list, tuple)) else
batch.shape[0]
    for key in step_metrics:
        metrics[key] += step_metrics[key] * batch_size
    num_elements += batch_size
metrics = {(log_prefix + key): (metrics[key] / num_elements).item() for key in
metrics}
return metrics

def is_new_model_better(self,
                       new_metrics: Dict[str, Any],
                       old_metrics: Dict[str, Any]) -> bool:
"""
Compares two sets of evaluation metrics to decide whether the
new model is better than the previous ones or not.

Args:
    new_metrics: A dictionary of the evaluation metrics of the new model.
    old_metrics: A dictionary of the evaluation metrics of the previously
    best model, i.e. the one to compare to.

Returns:

```

(continues on next page)

(continued from previous page)

```

    True if the new model is better than the old one, and False otherwise.
    """
    if old_metrics is None:
        return True
    for key, is_larger in [('val/val_metric', False), ('val/acc', True), ('val/loss',
→ False)]:
        if key in new_metrics:
            if is_larger:
                return new_metrics[key] > old_metrics[key]
            else:
                return new_metrics[key] < old_metrics[key]
    assert False, f'No known metrics to log on: {new_metrics}'


def tracker(self,
            iterator : Iterator,
            **kwargs) -> Iterator:
    """
    Wraps an iterator in a progress bar tracker (tqdm) if the progress bar
    is enabled.

    Args:
        iterator: Iterator to wrap in tqdm.
        kwargs: Additional arguments to tqdm.

    Returns:
        Wrapped iterator if progress bar is enabled, otherwise same iterator
        as input.
    """
    if self.enable_progress_bar:
        return tqdm(iterator, **kwargs)
    else:
        return iterator


def save_metrics(self,
                 filename : str,
                 metrics : Dict[str, Any]):
    """
    Saves a dictionary of metrics to file. Can be used as a textual
    representation of the validation performance for checking in the terminal.

    Args:
        filename: Name of the metrics file without folders and postfix.
        metrics: A dictionary of metrics to save in the file.
    """
    with open(os.path.join(self.log_dir, f'metrics/{filename}.json'), 'w') as f:
        json.dump(metrics, f, indent=4)


def on_training_start(self):
    """
    Method called before training is started. Can be used for additional
    initialization operations etc.
    """

```

(continues on next page)

(continued from previous page)

```

pass

def on_training_epoch_end(self,
                           epoch_idx : int):
    """
    Method called at the end of each training epoch. Can be used for additional
    logging or similar.

    Args:
        epoch_idx: Index of the training epoch that has finished.
    """
    pass

def on_validation_epoch_end(self,
                            epoch_idx : int,
                            eval_metrics : Dict[str, Any],
                            val_loader : Iterator):
    """
    Method called at the end of each validation epoch. Can be used for additional
    logging and evaluation.

    Args:
        epoch_idx: Index of the training epoch at which validation was performed.
        eval_metrics: A dictionary of the validation metrics. New metrics added to
                      this dictionary will be logged as well.
        val_loader: Data loader of the validation set, to support additional
                    evaluation.
    """
    pass

def save_model(self,
                 step : int = 0):
    """
    Saves current training state at certain training iteration. Only the model
    parameters and batch statistics are saved to reduce memory footprint. To
    support the training to be continued from a checkpoint, this method can be
    extended to include the optimizer state as well.

    Args:
        step: Index of the step to save the model at, e.g. epoch.
    """
    checkpoints.save_checkpoint(ckpt_dir=self.log_dir,
                               target={'params': self.state.params,
                                       'batch_stats': self.state.batch_stats},
                               step=step,
                               overwrite=True)

def load_model(self):
    """
    Loads model parameters and batch statistics from the logging directory.
    """
    state_dict = checkpoints.restore_checkpoint(ckpt_dir=self.log_dir, target=None)

```

(continues on next page)

(continued from previous page)

```

self.state = TrainState.create(apply_fn=self.model.apply,
                             params=state_dict['params'],
                             batch_stats=state_dict['batch_stats'],
                             # Optimizer will be overwritten when training
→ starts
→ sgd(0.1),
      rng=self.state.rng
    )

def bind_model(self):
    """
    Returns a model with parameters bound to it. Enables an easier inference
    access.

    Returns:
        The model with parameters and evt. batch statistics bound to it.
    """
    params = {'params': self.state.params}
    if self.state.batch_stats:
        params['batch_stats'] = self.state.batch_stats
    return self.model.bind(params)

@classmethod
def load_from_checkpoint(cls,
                         checkpoint : str,
                         exmp_input : Any) -> Any:
    """
    Creates a Trainer object with same hyperparameters and loaded model from
    a checkpoint directory.

    Args:
        checkpoint: Folder in which the checkpoint and hyperparameter file is stored.
        exmp_input: An input to the model for shape inference.

    Returns:
        A Trainer object with model loaded from the checkpoint folder.
    """
    hparams_file = os.path.join(checkpoint, 'hparams.json')
    assert os.path.isfile(hparams_file), 'Could not find hparams file'
    with open(hparams_file, 'r') as f:
        hparams = json.load(f)
    hparams.pop('model_class')
    hparams.update(hparams.pop('model_hparams'))
    if not hparams['logger_params']:
        hparams['logger_params'] = dict()
    hparams['logger_params']['log_dir'] = checkpoint
    trainer = cls(exmp_input=exmp_input,
                  **hparams)
    trainer.load_model()
    return trainer

```

Utility functions

Besides the Trainer module, we have seen other functionalities re-occurring several times in the tutorials. One of them is `numpy_collate`, which is needed for PyTorch's data loader to purely work with NumPy arrays. Similarly, creating Data Loaders for our datasets often follows the same structure, which we can also summarize in a function called `create_data_loaders`.

```
[4]: def numpy_collate(batch):
    if isinstance(batch[0], np.ndarray):
        return np.stack(batch)
    elif isinstance(batch[0], (tuple, list)):
        transposed = zip(*batch)
        return [numpy_collate(samples) for samples in transposed]
    else:
        return np.array(batch)

def create_data_loaders(*datasets : Sequence[data.Dataset],
                      train : Union[bool, Sequence[bool]] = True,
                      batch_size : int = 128,
                      num_workers : int = 4,
                      seed : int = 42):
    """
    Creates data loaders used in JAX for a set of datasets.

    Args:
        datasets: Datasets for which data loaders are created.
        train: Sequence indicating which datasets are used for
            training and which not. If single bool, the same value
            is used for all datasets.
        batch_size: Batch size to use in the data loaders.
        num_workers: Number of workers for each dataset.
        seed: Seed to initialize the workers and shuffling with.
    """
    loaders = []
    if not isinstance(train, (list, tuple)):
        train = [train for _ in datasets]
    for dataset, is_train in zip(datasets, train):
        loader = data.DataLoader(dataset,
                                batch_size=batch_size,
                                shuffle=is_train,
                                drop_last=is_train,
                                collate_fn=numpy_collate,
                                num_workers=num_workers,
                                persistent_workers=is_train,
                                generator=torch.Generator().manual_seed(seed))
        loaders.append(loader)
    return loaders
```

4.4.2 Example 1: Function regression

Using the `TrainerModule` and our few utility functions, we can now write a full training scenario with logging etc. in a few lines. To showcase this, we first consider a very simple scenario: regressing a sine-wave with a neural network.

Dataset

The first step is to create a dataset. Since we can use PyTorch's data package, this is straightforward. First, let's import needed plotting libraries for visualization and set the data and checkpoint path, similarly as in any other tutorial.

```
[5]: ## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()
sns.set()

DATASET_PATH = '../data/'
CHECKPOINT_PATH = '../saved_models/guide4/'
```

The dataset is kept very simple and contains pairs of input-output of a sine function:

```
[6]: def target_function(x):
    return np.sin(x * 3.0)

class RegressionDataset(data.Dataset):

    def __init__(self, num_points, seed):
        super().__init__()
        rng = np.random.default_rng(seed)
        self.x = rng.uniform(low=-2.0, high=2.0, size=num_points)
        self.y = target_function(self.x)

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        return self.x[idx:idx+1], self.y[idx:idx+1]
```

We can create our needed data loaders with the utility function `create_data_loaders` and visualize the dataset for debugging:

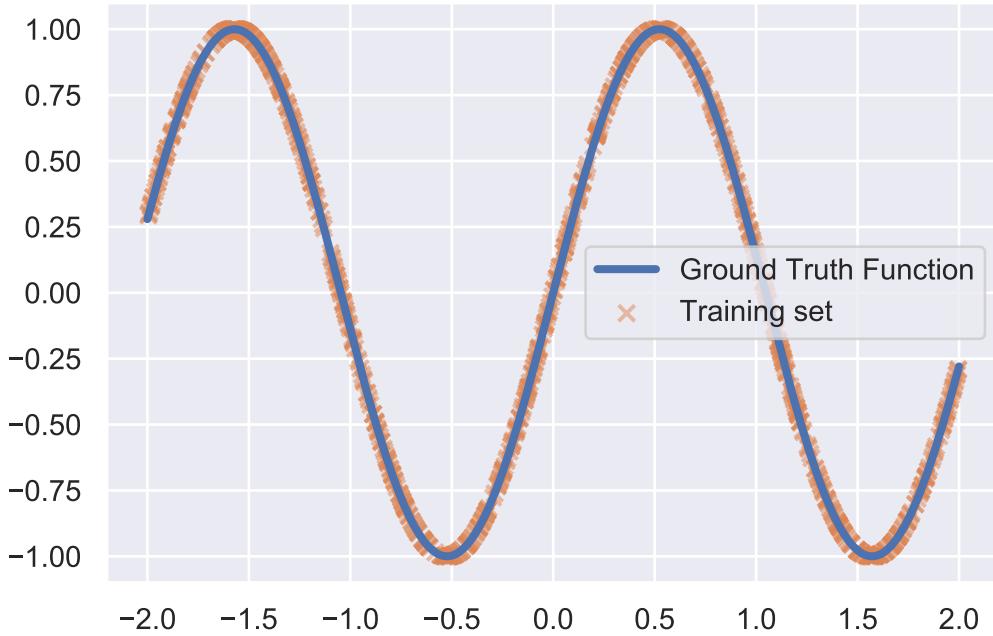
```
[7]: train_set = RegressionDataset(num_points=1000, seed=42)
val_set = RegressionDataset(num_points=200, seed=43)
test_set = RegressionDataset(num_points=500, seed=44)
train_loader, val_loader, test_loader = create_data_loaders(train_set, val_set, test_set,
                                                          train=[True, False, False],
                                                          batch_size=64)
```

(continues on next page)

(continued from previous page)

```
x = np.linspace(-2, 2, 1000)
plt.scatter(train_set.x, train_set.y, color='C1', marker='x', alpha=0.5, label='Training set')
plt.plot(x, target_function(x), linewidth=3.0, label='Ground Truth Function')
plt.legend()
plt.title('Regression function')
plt.show()
```

Regression function



Note that even with PyTorch Lightning, we would have needed the similar setup here.

Model

The Trainer module does not restrict us to any specific model, so let's implement our own MLP class for regression:

```
[8]: class MLPRegressor(nn.Module):
    hidden_dims : Sequence[int]
    output_dim : int

    @nn.compact
    def __call__(self, x, **kwargs):
        for dims in self.hidden_dims:
            x = nn.Dense(dims)(x)
            x = nn.silu(x)
        x = nn.Dense(self.output_dim)(x)
        return x
```

Trainer

Now comes the interesting part. Using the `TrainerModule`, we only need to overwrite the aspects that are needed: the training and validation step. For the regression task and the small model, this reduces to writing the mean-squared error loss. Note that we still have some minor repetitive code here, such as creating the gradient function with `jax.value_and_grad` in the training step and applying the update with `state.apply_gradients`. One could reduce this further by restricting possible loss functions, but to keep flexibility high, we implement the whole training step here.

```
[9]: class MLPRegressTrainer(TrainerModule):

    def __init__(self,
                 hidden_dims : Sequence[int],
                 output_dim : int,
                 **kwargs):
        super().__init__(model_class=MLPRegressor,
                        model_hparams={
                            'hidden_dims': hidden_dims,
                            'output_dim': output_dim
                        },
                        **kwargs)

    def create_functions(self):
        def mse_loss(params, batch):
            x, y = batch
            pred = self.model.apply({'params': params}, x)
            loss = ((pred - y) ** 2).mean()
            return loss

        def train_step(state, batch):
            loss_fn = lambda params: mse_loss(params, batch)
            loss, grads = jax.value_and_grad(loss_fn)(state.params)
            state = state.apply_gradients(grads=grads)
            metrics = {'loss': loss}
            return state, metrics

        def eval_step(state, batch):
            loss = mse_loss(state.params, batch)
            return {'loss': loss}

        return train_step, eval_step
```

And that's already it! This now looks much more like the minimal code of PyTorch Lightning and automatically logs our metrics as we want.

Training

To train the model, we simply specify our hyperparameters and create a Trainer module:

```
[10]: trainer = MLPRegressorTrainer(hidden_dims=[128, 128],
                                    output_dim=1,
                                    optimizer_hparams={'lr': 4e-3},
                                    logger_params={'base_log_dir': CHECKPOINT_PATH},
                                    example_input=next(iterator(train_loader))[0:1],
                                    check_val_every_n_epoch=5)
```

MLPRegressor Summary			
path	outputs	params	
Inputs	- float64[64,1] - train: True		
Dense_0	float32[64,128]	bias: float32[128] kernel: float32[1,128] 256 (1.0 KB)	
Dense_1	float32[64,128]	bias: float32[128] kernel: float32[128,128] 16,512 (66.0 KB)	
Dense_2	float32[64,1]	bias: float32[1] kernel: float32[128,1] 129 (516 B)	
MLPRegressor	float32[64,1]		
	Total	16,897 (67.6 KB)	

Total Parameters: 16,897 (67.6 KB)

As one can see, we also automatically print out all layers with their parameters and outputs with Flax's `nn.tabulate` function. This is quite helpful for debugging and gives an intuition about the size of the model. Since the task is not very difficult, we are fine with using less than 20k parameters.

Next, let's start the training:

```
[11]: metrics = trainer.train_model(train_loader,
                                    val_loader,
                                    test_loader=test_loader,
                                    num_epochs=50)
```

```
[12]: print(f'Training loss: {metrics["train/loss"]}')
print(f'Validation loss: {metrics["val/loss"]}')
print(f'Test loss: {metrics["test/loss"]}'')
```

Training loss: 0.0008829445578157902
 Validation loss: 0.0008724514045752585
 Test loss: 0.0007670423365198076

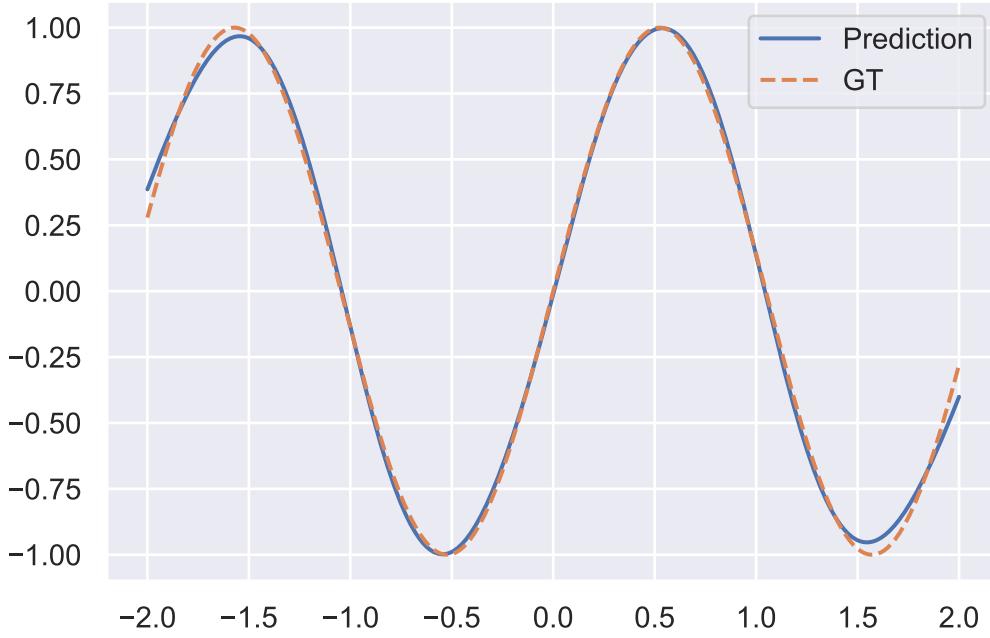
With the speed of JAX, this takes only a few seconds. The logs can be found in `../saved_models/guide4/MLPRegressor/`, and we have a dictionary of the best results here as well. The training, validation and test loss suggest that our model learned the function quite well, but let's check it by predicting the whole function as inference task.

Inference

To perform inference, we first bind the model to the parameters. This enables us a simpler API, closer to PyTorch. Applying the model to a values between -2.0 and 2.0 shows that the model learned the sine wave indeed quite well and is only off at the corners.

```
[13]: model_bd = trainer.bind_model()
x = np.linspace(-2, 2, 1000)[:,None]
y_pred = model_bd(x)
plt.plot(x, y_pred, label='Prediction')
plt.plot(x, target_function(x), '--', label='GT')
plt.title('Function regression')
plt.legend()
plt.show()
```

Function regression



4.4.3 Example 2: CIFAR10 classification

As a second example, let's consider image classification on CIFAR10. We have done it before in [Tutorial 5](#), but here, we want to showcase the flexibility of the Trainer module. For that, we will consider a more complicated setting in Flax: a model with both Batch Normalization and Dropout. However, with the Trainer module, this reduces to a simpler code again.

Dataset

First, let's load our dataset again. This is the same data loading as used in [Tutorial 5](#) and [Tutorial 15](#), but only considers flipping as regularization technique since we work with simple MLPs here.

```
[14]: from torchvision.datasets import CIFAR10
from torchvision import transforms

# Transformations applied on each image => bring them into a numpy array
DATA_MEANS = np.array([0.49139968, 0.48215841, 0.44653091])
DATA_STD = np.array([0.24703223, 0.24348513, 0.26158784])
def image_to_numpy(img):
    img = np.array(img, dtype=np.float32)
    img = (img / 255. - DATA_MEANS) / DATA_STD
    return img

test_transform = image_to_numpy
# For training, we add some augmentation. Networks are too powerful and would overfit.
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                      image_to_numpy])
# Loading the training dataset. We need to split it into a training and validation part
# We need to do a little trick because the validation set should not use the
# augmentation.
train_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=train_transform,
                        download=True)
val_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=test_transform,
                      download=True)
train_set, _ = data.random_split(train_dataset, [45000, 5000], generator=torch.
                                 Generator().manual_seed(42))
_, val_set = data.random_split(val_dataset, [45000, 5000], generator=torch.Generator().
                                manual_seed(42))

# Loading the test set
test_set = CIFAR10(root=DATASET_PATH, train=False, transform=test_transform,
                    download=True)

train_loader, val_loader, test_loader = create_data_loaders(train_set, val_set, test_set,
                                                            train=[True, False, False],
                                                            batch_size=256)
```

Files already downloaded and verified
 Files already downloaded and verified
 Files already downloaded and verified

Model

The model definition is again relatively simple. We repeat a series of Dropout → Linear → BatchNorm → Swish blocks, with a final Dropout and Linear layer at the end.

```
[15]: class MLPClassifier(nn.Module):
    hidden_dims : Sequence[int]
    num_classes : int
    dropout_prob : float = 0.0

    @nn.compact
    def __call__(self, x, train=True):
        x = x.reshape(x.shape[0], -1)
        for dims in self.hidden_dims:
            x = nn.Dropout(self.dropout_prob)(x, deterministic=not train)
            x = nn.Dense(dims)(x)
            x = nn.BatchNorm()(x, use_running_average=not train)
            x = nn.swish(x)
        x = nn.Dropout(self.dropout_prob)(x, deterministic=not train)
        x = nn.Dense(self.num_classes)(x)
        return x
```

Trainer

For the Trainer module, we again define our model hyperparameters in the init function, and write our own training and evaluation steps in `create_functions`. In these functions, we take care of the mutable batch statistics and the PRNG state for dropout. Note that since both parts are integrated in the `TrainState`, we do not need to alternate the training or validation step signature, and it is sufficient to pass the state and batch to the functions. Additionally, we overwrite the model call during initialization (`run_model_init`) and tabulate function (`print_tabulate`). And that's it! Overall, we didn't need to make many changes, showing that the trainer module is flexible enough to support a variety of layers. For now, we can ignore the `trial` object and come back to it later when discussing automated hyperparameter tuning.

```
[16]: class MLPClassTrainer(TrainerModule):

    def __init__(self,
                 hidden_dims : Sequence[int],
                 num_classes : int,
                 dropout_prob : float,
                 trial : Any = None,
                 **kwargs):
        super().__init__(model_class=MLPClassifier,
                        model_hparams={
                            'hidden_dims': hidden_dims,
                            'num_classes': num_classes,
                            'dropout_prob': dropout_prob
                        },
                        **kwargs)
        self.trial = trial

    def create_functions(self):
        def loss_function(params, batch_stats, rng, batch, train):
            ...
```

(continues on next page)

(continued from previous page)

```

        imgs, labels = batch
        rng, dropout_rng = random.split(rng)
        output = self.model.apply({'params': params, 'batch_stats': batch_stats},
                                  imgs,
                                  train=train,
                                  rands={'dropout': dropout_rng},
                                  mutable=['batch_stats'] if train else False)
        logits, new_model_state = output if train else (output, None)
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, labels).mean()
        acc = (logits.argmax(axis=-1) == labels).mean()
        return loss, (rng, new_model_state, acc)

    def train_step(state, batch):
        loss_fn = lambda params: loss_function(params, state.batch_stats, state.rng, batch, train=True)
        ret, grads = jax.value_and_grad(loss_fn, has_aux=True)(state.params)
        loss, rng, new_model_state, acc = ret[0], *ret[1]
        state = state.apply_gradients(grads=grads, batch_stats=new_model_state['batch_stats'], rng=rng)
        metrics = {'loss': loss, 'acc': acc}
        return state, metrics

    def eval_step(state, batch):
        _, _, acc = loss_function(state.params, state.batch_stats, state.rng, batch, train=False)
        return {'acc': acc}

    return train_step, eval_step

def run_model_init(self, exmp_input, init_rng):
    imgs, _ = exmp_input
    init_rng, dropout_rng = random.split(init_rng)
    return self.model.init({'params': init_rng, 'dropout': dropout_rng}, x=imgs, train=True)

def print_tabulate(self, exmp_input):
    imgs, _ = exmp_input
    print(self.model.tabulate(rands={'params': random.PRNGKey(0), 'dropout': random.PRNGKey(0)}, x=imgs, train=True))

def on_validation_epoch_end(self, epoch_idx, eval_metrics, val_loader):
    if self.trial:
        self.trial.report(eval_metrics['val/acc'], step=epoch_idx)
        if self.trial.should_prune():
            raise optuna.exceptions.TrialPruned()

```

Training

With the Trainer fully defined, we can again start training. Let's pick some reasonable hyperparameters, and look at the layers created by the model:

[17]:	trainer = MLPClassTrainer(hidden_dims=[512, 512], num_classes=10, dropout_prob=0.4, optimizer_hparams={ 'weight_decay': 2e-4, 'lr': 1e-3 }, logger_params={ 'base_log_dir': CHECKPOINT_PATH }, exmp_input=next(iter(train_loader)), check_val_every_n_epoch=5)		
MLPClassifier Summary			
path	outputs	batch_stats	params
Inputs	train: True x: float64[256,32,32,3]		
BatchNorm_0	float32[256,512] var: float32[512] scale: float32[512] 1,024 (4.1 KB)	mean: float32[512] bias: float32[512] 1,024 (4.1 KB)	
BatchNorm_1	float32[256,512] var: float32[512] scale: float32[512] 1,024 (4.1 KB)	mean: float32[512] bias: float32[512] 1,024 (4.1 KB)	
Dense_0	float32[256,512] kernel: float32[3072, 512] 1,573,376 (6.3 MB)	bias: float32[512] kernel: float32[3072, 512] 1,573,376 (6.3 MB)	

(continues on next page)

(continued from previous page)

Dense_1	float32[256, 512]		bias: float32[512]	[¶]
			kernel: float32[512,	[¶]
				[¶]
			262,656 (1.1 MB)	[¶]
				[¶]
Dense_2	float32[256, 10]		bias: float32[10]	[¶]
			kernel: float32[512, 10]	[¶]
				[¶]
			5,130 (20.5 KB)	[¶]
				[¶]
Dropout_0	float32[256, 3072]			[¶]
				[¶]
Dropout_1	float32[256, 512]			[¶]
				[¶]
Dropout_2	float32[256, 512]			[¶]
				[¶]
MLPClassifier	float32[256, 10]			[¶]
				[¶]
		Total 2,048 (8.2 KB)	1,843,210 (7.4 MB)	[¶]
				[¶]
		Total Parameters: 1,845,258 (7.4 MB)		[¶]
				[¶]

One interesting observation here is that the MLP has way more parameters than any of the CNNs in [Tutorial 5](#), but yet significantly underperforms the models. Although to really see the performance, let's train the model again with the simple call from before:

```
[18]: metrics = trainer.train_model(train_loader,
                                    val_loader,
                                    test_loader=test_loader,
                                    num_epochs=50)
```

```
[19]: print(f'Validation accuracy: {metrics["val/acc"] :4.2%}')
print(f'Test accuracy: {metrics["test/acc"] :4.2%}')

Validation accuracy: 60.22%
Test accuracy: 60.05%
```

The MLP achieves decent accuracy, but already for models like this, we have several hyperparameters to tune including learning rate, weight decay, dropout rate. Which should we choose? While we could use intuition to get a reasonable guess, it is unlikely that we hit the best hyperparameter set. In order to find a very strong hyperparameter set, a good practice is to use automatic hyperparameter tuning, which we shortly review next to showcase the flexibility of the Trainer module.

4.4.4 Automatic hyperparameter tuning with Optuna

Automatic hyperparameter tuner have the goal to efficiently identify sets of hyperparameters that achieve the best performance. Thereby, the key question is how can we search the hyperparameter space efficiently, since we don't have infinite compute. [Optuna](#) is a library that helps you setup this search with minimal code overhead and perform automatic hyperparameter tuning. Before getting started with Optuna, let's import the library and download a pre-executed hyperparameter search as an example.

```
[20]: try:
    import optuna
except ModuleNotFoundError:
    !pip install --quiet --upgrade optuna pyplot
    import optuna
```

```
[21]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/guide4/"
# Files to download
pretrained_files = ["optuna_hparam_search.db", "MLPClassifier/version_16/checkpoint_150",
                    "MLPClassifier/version_16/hparams.json"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please contact the author with the full output",
                  "including the following error:\n", e)
```

Defining objective and hyperparameters

The main part a user has to specify in Optuna is intuitively the the objective to optimize, and the hyperparameters over which we want to optimize. In our case, the objective is to optimize the validation accuracy of the MLP. Note that we do not use the test set here, since hyperparameter searches should only be done on the validation set, not the “unseen” test set! The function below, `objective(trial)`, creates a MLP with our trainer module and trains it for max. 200 epochs. The input argument, `trial`, is thereby an object which characterizes the current run. This includes, for example, the hyperparameters we want to optimize. In order to add a hyperparameter to our optimization set, we can simply call `trial.suggest_float` for continuous values and `trial.suggest_categorical` for categorical values (e.g. which optimizer to use). For the CIFAR10 classification, we consider the following three hyperparameters: dropout rate, weight decay, and the learning rate, which we define below. Finally, we return the best validation accuracy which will be used by Optuna to guide the next pick of hyperparameters.

```
[22]: def objective(trial):
    my_train_loader, my_val_loader = create_data_loaders(train_set, val_set,
                                                          train=[True, False],
                                                          batch_size=256)
    trainer = MLPClassTrainer(hidden_dims=[512, 512],
                               num_classes=10,
                               dropout_prob=trial.suggest_float('dropout_prob', 0, 0.6),
                               optimizer_hparams={
                                   'weight_decay': trial.suggest_float('weight_decay', 1e-6, 1e-2, log=True),
                                   'lr': trial.suggest_float('lr', 1e-4, 1e-2, log=True)
                               },
                               logger_params={
                                   'base_log_dir': CHECKPOINT_PATH
                               },
                               exmp_input=next(iter(my_train_loader)),
                               check_val_every_n_epoch=5,
                               trial=trial)
    metrics = trainer.train_model(my_train_loader,
                                  my_val_loader,
                                  num_epochs=200)
    del trainer
    del my_train_loader, my_val_loader
    return metrics['val/acc']
```

Running hyperparameter study

To run the hyperparameter search, we create a Study in Optuna. A study implements the search logic and summarizes the data/logs of all executed experiments. By default, Optuna uses the [Tree-Structured Parzen Estimator](#) algorithm, using Gaussian Mixture Models to estimate the performance surface of they hyperparameters. For more information, check out the [documentation](#). The studies are usually stored in a database format, but can be easily accessed via the Python interface of Optuna. Let’s run the hyperparameter search for up to 25 models:

```
[23]: study = optuna.create_study(
    study_name='mlp_cifar10',
    storage=f'sqlite:///{CHECKPOINT_PATH}/optuna_hparam_search.db',
    direction='maximize',
    pruner=optuna.pruners.MedianPruner(n_startup_trials=5, n_warmup_steps=50),
    load_if_exists=True
```

(continues on next page)

(continued from previous page)

```
)
study.optimize(objective, n_trials=25-len(study.trials), n_jobs=1)

[I 2022-07-05 20:06:47,966] Using an existing study with name 'mlp_cifar10' instead of ↵
creating a new one.
```

During the study creation, we used the input argument `pruner`. This specifies a strategy with which we want to stop experiments early if they don't look promising. For instance, a very low learning rate combined with high weight decay and dropout will likely achieve low performance, which we can already judge after 50 epochs and don't have to run the model for much longer. For this, we implemented the `on_validation_epoch_end` callback in our Trainer module before. After each epoch, it reports the current validation performance to Optuna. Depending on the previous performances and Optuna's pruning strategy, it may decide to stop the experiment early, which it does by throwing a `TrialPruned` error. This error is caught by Optuna, and the next trial is directly started.

Evaluate hyperparameter search

After finishing the hyperparameter search, we can analyze the results. First, let's print the best model found and its corresponding hyperparameters:

```
[24]: trial = study.best_trial
print(f'Best Validation Accuracy: {trial.value:.2%}')
print(f'Best Params:')
for key, value in trial.params.items():
    print(f'-> {key}: {value}')


Best Validation Accuracy: 63.44%
Best Params:
-> dropout_prob: 0.39573629692783413
-> lr: 0.002097404408052793
-> weight_decay: 0.0012107132860246818
```

The validation performance is quite a bit higher than the model we had manually designed before. Let's load the model and check its test performance. For this, we can make use of the `load_from_checkpoint` function of our Trainer module:

```
[25]: trainer = MLPClassTrainer.load_from_checkpoint(os.path.join(CHECKPOINT_PATH,
    'MLPClassifier/version_16/'),
    exmp_input=next(iter(train_loader)))
```

```
test_metrics = trainer.eval_model(test_loader)
print(f'Test accuracy: {test_metrics["acc"]:.2%}')


MLPClassifier Summary
```

path	outputs	batch_stats	params
Inputs	train: True		
	x: float64[256,32,32,3]		
BatchNorm_0	float32[256,512]	mean: float32[512] bias: float32[512]	
		var: float32[512] scale: float32[512]	

(continues on next page)

(continued from previous page)

			1,024 (4.1 KB)	1,024 (4.1 KB)
BatchNorm_1	float32[256,512]	mean: float32[512]	bias: float32[512]	
		var: float32[512]	scale: float32[512]	
		1,024 (4.1 KB)	1,024 (4.1 KB)	
Dense_0	float32[256,512]		bias: float32[512]	
512]			kernel: float32[3072,	
			1,573,376 (6.3 MB)	
Dense_1	float32[256,512]		bias: float32[512]	
512]			kernel: float32[512,	
			262,656 (1.1 MB)	
Dense_2	float32[256,10]		bias: float32[10]	
			kernel: float32[512,10]	
			5,130 (20.5 KB)	
Dropout_0	float32[256,3072]			
Dropout_1	float32[256,512]			
Dropout_2	float32[256,512]			
MLPClassifier	float32[256,10]			

(continues on next page)

(continued from previous page)

	Total 2,048 (8.2 KB)	1,843,210 (7.4 MB)	
Total Parameters: 1,845,258 (7.4 MB)			
Test accuracy: 62.89%			

The test performance is also quite strong, showing the benefit of the automatic hyperparameter search. However, often, we are not just interested in the best model. Optuna provides several ways of visualizing the results of the hyperparameter study, for instance by plotting all validation accuracy curves:

```
[26]: fig = optuna.visualization.plot_intermediate_values(study)
fig.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

As we can see, there is quite a big group of models that perform very similarly. At the same time, there were a few models with very poor results, which were lucky stopped early by Optuna to not waste compute.

Another question we might have is which hyperparameter was the most important for the performance? This can be directly visualized by `plot_param_importances`. By default, Optuna uses a random forest regression to estimate the importance of each hyperparameter ([documentation](#)), and shows it as values that sum up to 1 for all hyperparameters:

```
[27]: fig = optuna.visualization.plot_param_importances(study)
fig.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Interestingly, the learning rate is the most important hyperparameter.

Finally, to get a good intuition of the interplay between hyperparameters, we can plot the estimated accuracy surface over hyperparameters:

```
[28]: fig = optuna.visualization.plot_contour(study, params=['lr', 'dropout_prob'])
fig.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

As the hyperparameter importance plot already suggested, the learning rate seems to be very important while we are more flexible in the dropout probability. However, there seems to be an optimal spot in the middle, overall maximizing the accuracy.

4.4.5 Conclusion

In this guide, we discussed tips and practices for doing research with JAX. We mainly focused on implementing a Trainer module that summarizes most common functionalities needed for training and testing models. To showcase the flexibility of the module, we implement a regression and classification model in a few lines. Moreover, we show how one can easily perform automatic hyperparameter optimization with Optuna in this setting. While this guide gives a possible template for a research code, it may not cover all parts that are needed for your specific usecase. Moreover, if you are specialized on a certain task, such as classification or generative models, you can further specialize the Trainer module to reduce your code for submodules.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.5 Tutorial 2: Introduction to PyTorch

Filled notebook:

Recordings:

Author: Phillip Lippe

Welcome to our PyTorch tutorial for the Deep Learning course 2022 at the University of Amsterdam! The following notebook is meant to give a short introduction to PyTorch basics, and get you setup for writing your own neural networks. PyTorch is an open source machine learning framework that allows you to write your own neural networks and optimize them efficiently. However, PyTorch is not the only framework of its kind. Alternatives to PyTorch include [TensorFlow](#), [JAX](#) and [Caffe](#). We choose to teach PyTorch at the University of Amsterdam because it is well established, has a huge developer community (originally developed by Facebook), is very flexible and especially used in research. Many current papers publish their code in PyTorch, and thus it is good to be familiar with PyTorch as well. Meanwhile, TensorFlow (developed by Google) is usually known for being a production-grade deep learning library. Still, if you know one machine learning framework in depth, it is very easy to learn another one because many of them use the same concepts and ideas. For instance, TensorFlow's version 2 was heavily inspired by the most popular features of PyTorch, making the frameworks even more similar. If you are already familiar with PyTorch and have created your own neural network projects, feel free to just skim this notebook.

We are of course not the first ones to create a PyTorch tutorial. There are many great tutorials online, including the “60-min blitz” on the official [PyTorch website](#). Yet, we choose to create our own tutorial which is designed to give you the basics particularly necessary for the practicals, but still understand how PyTorch works under the hood. Over the next few weeks, we will also keep exploring new PyTorch features in the series of Jupyter notebook tutorials about deep learning.

We will use a set of standard libraries that are often used in machine learning projects. If you are running this notebook on Google Colab, all libraries should be pre-installed. If you are running this notebook locally, make sure you have installed our `dl2022` environment and have activated it.

```
[1]: ## Standard libraries
import os
import math
import numpy as np
import time

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgba
import seaborn as sns
sns.set()

## Progress bar
from tqdm.notebook import tqdm

/tmp/ipykernel_875535/3937499902.py:11: DeprecationWarning: `set_matplotlib_formats` is
  deprecated since IPython 7.23, directly use `matplotlib_inline.backend_inline.set_
  matplotlib_formats()`
    set_matplotlib_formats('svg', 'pdf') # For export
```

4.5.1 The Basics of PyTorch

We will start with reviewing the very basic concepts of PyTorch. As a prerequisite, we recommend to be familiar with the `numpy` package as most machine learning frameworks are based on very similar concepts. If you are not familiar with `numpy` yet, don't worry: here is a [tutorial](#) to go through.

So, let's start with importing PyTorch. The package is called `torch`, based on its original framework `Torch`. As a first step, we can check its version:

```
[2]: import torch
print("Using torch", torch.__version__)

Using torch 1.13.0
```

At the time of writing this tutorial (end of October 2022), the current stable version is 1.13. You should therefore see the output `Using torch 1.13.0` or `Using torch 1.12.0`, eventually with some extension for the CUDA version on Colab. In case you use the `dl2022` environment, you should see `Using torch 1.13.0`. In general, it is recommended to keep the PyTorch version updated to the newest one. If you see a lower version number than 1.12, make sure you have installed the correct environment, or ask one of your TAs. In case PyTorch 1.14 or newer will be published during the time of the course, don't worry. The interface between PyTorch versions doesn't change too much, and hence all code should also be runnable with newer versions.

As in every machine learning framework, PyTorch provides functions that are stochastic like generating random numbers. However, a very good practice is to setup your code to be reproducible with the exact same random numbers. This is why we set a seed below.

```
[3]: torch.manual_seed(42) # Setting the seed
[3]: <torch._C.Generator at 0x7fa327d6da90>
```

Tensors

Tensors are the PyTorch equivalent to Numpy arrays, with the addition to also have support for GPU acceleration (more on that later). The name “tensor” is a generalization of concepts you already know. For instance, a vector is a 1-D tensor, and a matrix a 2-D tensor. When working with neural networks, we will use tensors of various shapes and number of dimensions.

Most common functions you know from numpy can be used on tensors as well. Actually, since numpy arrays are so similar to tensors, we can convert most tensors to numpy arrays (and back) but we don't need it too often.

Initialization

Let's first start by looking at different ways of creating a tensor. There are many possible options, the simplest one is to call `torch.Tensor` passing the desired shape as input argument:

```
[4]: x = torch.Tensor(2, 3, 4)
print(x)

tensor([[[ 6.5254e+10,  3.0890e-41,  4.2039e-45, -6.3663e-15],
        [ 6.5246e+10,  3.0890e-41,  1.6367e-42,  4.5787e-41],
        [ 6.5254e+10,  3.0890e-41,  4.2039e-45,  4.5787e-41]],

       [[ 6.5255e+10,  3.0890e-41,  1.4013e-45,  0.0000e+00],
        [ 6.5255e+10,  3.0890e-41,  6.5246e+10,  3.0890e-41],
        [ 1.7404e-42, -5.0820e+26,  6.5255e+10,  3.0890e-41]])
```

The function `torch.Tensor` allocates memory for the desired tensor, but reuses any values that have already been in the memory. To directly assign values to the tensor during initialization, there are many alternatives including:

- `torch.zeros`: Creates a tensor filled with zeros
- `torch.ones`: Creates a tensor filled with ones
- `torch.rand`: Creates a tensor with random values uniformly sampled between 0 and 1
- `torch.randn`: Creates a tensor with random values sampled from a normal distribution with mean 0 and variance 1
- `torch.arange`: Creates a tensor containing the values $N, N + 1, N + 2, \dots, M$
- `torch.Tensor` (input list): Creates a tensor from the list elements you provide

```
[5]: # Create a tensor from a (nested) list
x = torch.Tensor([[1, 2], [3, 4]])
print(x)

tensor([[1., 2.],
        [3., 4.]])
```

```
[6]: # Create a tensor with random values between 0 and 1 with the shape [2, 3, 4]
x = torch.rand(2, 3, 4)
print(x)

tensor([[[0.8823, 0.9150, 0.3829, 0.9593],
         [0.3904, 0.6009, 0.2566, 0.7936],
         [0.9408, 0.1332, 0.9346, 0.5936]],
```

(continues on next page)

(continued from previous page)

```
[[0.8694, 0.5677, 0.7411, 0.4294],  
 [0.8854, 0.5739, 0.2666, 0.6274],  
 [0.2696, 0.4414, 0.2969, 0.8317]]])
```

You can obtain the shape of a tensor in the same way as in numpy (`x.shape`), or using the `.size` method:

```
[7]: shape = x.shape  
print("Shape:", x.shape)  
  
size = x.size()  
print("Size:", size)  
  
dim1, dim2, dim3 = x.size()  
print("Size:", dim1, dim2, dim3)  
  
Shape: torch.Size([2, 3, 4])  
Size: torch.Size([2, 3, 4])  
Size: 2 3 4
```

Tensor to Numpy, and Numpy to Tensor

Tensors can be converted to numpy arrays, and numpy arrays back to tensors. To transform a numpy array into a tensor, we can use the function `torch.from_numpy`:

```
[8]: np_arr = np.array([[1, 2], [3, 4]])  
tensor = torch.from_numpy(np_arr)  
  
print("Numpy array:", np_arr)  
print("PyTorch tensor:", tensor)  
  
Numpy array: [[1 2]  
 [3 4]]  
PyTorch tensor: tensor([[1, 2],  
 [3, 4]])
```

To transform a PyTorch tensor back to a numpy array, we can use the function `.numpy()` on tensors:

```
[9]: tensor = torch.arange(4)  
np_arr = tensor.numpy()  
  
print("PyTorch tensor:", tensor)  
print("Numpy array:", np_arr)  
  
PyTorch tensor: tensor([0, 1, 2, 3])  
Numpy array: [0 1 2 3]
```

The conversion of tensors to numpy require the tensor to be on the CPU, and not the GPU (more on GPU support in a later section). In case you have a tensor on GPU, you need to call `.cpu()` on the tensor beforehand. Hence, you get a line like `np_arr = tensor.cpu().numpy()`.

Operations

Most operations that exist in numpy, also exist in PyTorch. A full list of operations can be found in the PyTorch documentation, but we will review the most important ones here.

The simplest operation is to add two tensors:

```
[10]: x1 = torch.rand(2, 3)
x2 = torch.rand(2, 3)
y = x1 + x2

print("X1", x1)
print("X2", x2)
print("Y", y)

X1 tensor([[0.1053, 0.2695, 0.3588],
           [0.1994, 0.5472, 0.0062]])
X2 tensor([[0.9516, 0.0753, 0.8860],
           [0.5832, 0.3376, 0.8090]])
Y tensor([[1.0569, 0.3448, 1.2448],
           [0.7826, 0.8848, 0.8151]])
```

Calling `x1 + x2` creates a new tensor containing the sum of the two inputs. However, we can also use in-place operations that are applied directly on the memory of a tensor. We therefore change the values of `x2` without the chance to re-accessing the values of `x2` before the operation. An example is shown below:

```
[11]: x1 = torch.rand(2, 3)
x2 = torch.rand(2, 3)
print("X1 (before)", x1)
print("X2 (before)", x2)

x2.add_(x1)
print("X1 (after)", x1)
print("X2 (after)", x2)

X1 (before) tensor([[0.5779, 0.9040, 0.5547],
                     [0.3423, 0.6343, 0.3644]])
X2 (before) tensor([[0.7104, 0.9464, 0.7890],
                     [0.2814, 0.7886, 0.5895]])
X1 (after) tensor([[0.5779, 0.9040, 0.5547],
                     [0.3423, 0.6343, 0.3644]])
X2 (after) tensor([[1.2884, 1.8504, 1.3437],
                     [0.6237, 1.4230, 0.9539]])
```

In-place operations are usually marked with a underscore postfix (e.g. “`add_`” instead of “`add`”).

Another common operation aims at changing the shape of a tensor. A tensor of size (2,3) can be re-organized to any other shape with the same number of elements (e.g. a tensor of size (6), or (3,2), ...). In PyTorch, this operation is called `view`:

```
[12]: x = torch.arange(6)
print("X", x)

X tensor([0, 1, 2, 3, 4, 5])
```

```
[13]: x = x.view(2, 3)
print("X", x)

X tensor([[0, 1, 2],
           [3, 4, 5]])
```

```
[14]: x = x.permute(1, 0) # Swapping dimension 0 and 1
print("X", x)

X tensor([[0, 3],
           [1, 4],
           [2, 5]])
```

Other commonly used operations include matrix multiplications, which are essential for neural networks. Quite often, we have an input vector \mathbf{x} , which is transformed using a learned weight matrix \mathbf{W} . There are multiple ways and functions to perform matrix multiplication, some of which we list below:

- `torch.matmul`: Performs the matrix product over two tensors, where the specific behavior depends on the dimensions. If both inputs are matrices (2-dimensional tensors), it performs the standard matrix product. For higher dimensional inputs, the function supports broadcasting (for details see the [documentation](#)). Can also be written as $\mathbf{a} @ \mathbf{b}$, similar to numpy.
- `torch.mm`: Performs the matrix product over two matrices, but doesn't support broadcasting (see [documentation](#))
- `torch.bmm`: Performs the matrix product with a support batch dimension. If the first tensor T is of shape $(b \times n \times m)$, and the second tensor R $(b \times m \times p)$, the output O is of shape $(b \times n \times p)$, and has been calculated by performing b matrix multiplications of the submatrices of T and R : $O_i = T_i @ R_i$
- `torch.einsum`: Performs matrix multiplications and more (i.e. sums of products) using the Einstein summation convention. Explanation of the Einstein sum can be found in assignment 1.

Usually, we use `torch.matmul` or `torch.bmm`. We can try a matrix multiplication with `torch.matmul` below.

```
[15]: x = torch.arange(6)
x = x.view(2, 3)
print("X", x)

X tensor([[0, 1, 2],
           [3, 4, 5]])
```

```
[16]: W = torch.arange(9).view(3, 3) # We can also stack multiple operations in a single line
print("W", W)

W tensor([[0, 1, 2],
           [3, 4, 5],
           [6, 7, 8]])
```

```
[17]: h = torch.matmul(x, W) # Verify the result by calculating it by hand too!
print("h", h)

h tensor([[15, 18, 21],
           [42, 54, 66]])
```

Indexing

We often have the situation where we need to select a part of a tensor. Indexing works just like in numpy, so let's try it:

```
[18]: x = torch.arange(12).view(3, 4)
print("X", x)
```

```
X tensor([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11]])
```

```
[19]: print(x[:, 1])    # Second column
tensor([1, 5, 9])
```

```
[20]: print(x[0])      # First row
tensor([0, 1, 2, 3])
```

```
[21]: print(x[:2, -1]) # First two rows, last column
tensor([3, 7])
```

```
[22]: print(x[1:3, :]) # Middle two rows
tensor([[ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

Dynamic Computation Graph and Backpropagation

One of the main reasons for using PyTorch in Deep Learning projects is that we can automatically get **gradients/derivatives** of functions that we define. We will mainly use PyTorch for implementing neural networks, and they are just fancy functions. If we use weight matrices in our function that we want to learn, then those are called the **parameters** or simply the **weights**.

If our neural network would output a single scalar value, we would talk about taking the **derivative**, but you will see that quite often we will have **multiple** output variables ("values"); in that case we talk about **gradients**. It's a more general term.

Given an input **x**, we define our function by **manipulating** that input, usually by matrix-multiplications with weight matrices and additions with so-called bias vectors. As we manipulate our input, we are automatically creating a **computational graph**. This graph shows how to arrive at our output from our input. PyTorch is a **define-by-run** framework; this means that we can just do our manipulations, and PyTorch will keep track of that graph for us. Thus, we create a dynamic computation graph along the way.

So, to recap: the only thing we have to do is to compute the **output**, and then we can ask PyTorch to automatically get the **gradients**.

Note: Why do we want gradients? Consider that we have defined a function, a neural net, that is supposed to compute a certain output y for an input vector x . We then define an **error measure** that tells us how wrong our network is; how bad it is in predicting output y from input x . Based on this error measure, we can use the gradients to **update** the weights W that were responsible for the output, so that the next time we present input x to our network, the output will be closer to what we want.

The first thing we have to do is to specify which tensors require gradients. By default, when we create a tensor, it does not require gradients.

```
[23]: x = torch.ones((3,))
print(x.requires_grad)

False
```

We can change this for an existing tensor using the function `requires_grad_()` (underscore indicating that this is an in-place operation). Alternatively, when creating a tensor, you can pass the argument `requires_grad=True` to most initializers we have seen above.

```
[24]: x.requires_grad_(True)
print(x.requires_grad)

True
```

In order to get familiar with the concept of a computation graph, we will create one for the following function:

$$y = \frac{1}{|x|} \sum_i [(x_i + 2)^2 + 3]$$

You could imagine that x are our parameters, and we want to optimize (either maximize or minimize) the output y . For this, we want to obtain the gradients $\partial y / \partial x$. For our example, we'll use $x = [0, 1, 2]$ as our input.

```
[25]: x = torch.arange(3, dtype=torch.float32, requires_grad=True) # Only float tensors can
           ↳ have gradients
print("X", x)

X tensor([0., 1., 2.], requires_grad=True)
```

Now let's build the computation graph step by step. You can combine multiple operations in a single line, but we will separate them here to get a better understanding of how each operation is added to the computation graph.

```
[26]: a = x + 2
b = a ** 2
c = b + 3
y = c.mean()
print("Y", y)

Y tensor(12.6667, grad_fn=<MeanBackward0>)
```

Using the statements above, we have created a computation graph that looks similar to the figure below:

We calculate a based on the inputs x and the constant 2, b is a squared, and so on. The visualization is an abstraction of the dependencies between inputs and outputs of the operations we have applied. Each node of the computation graph has automatically defined a function for calculating the gradients with respect to its inputs, `grad_fn`. You can see this when we printed the output tensor y . This is why the computation graph is usually visualized in the reverse direction (arrows point from the result to the inputs). We can perform backpropagation on the computation graph by calling the function `backward()` on the last output, which effectively calculates the gradients for each tensor that has the property `requires_grad=True`:

```
[27]: y.backward()
```

$x.grad$ will now contain the gradient $\partial y / \partial x$, and this gradient indicates how a change in x will affect output y given the current input $x = [0, 1, 2]$:

[28]: `print(x.grad)`

```
tensor([1.3333, 2.0000, 2.6667])
```

We can also verify these gradients by hand. We will calculate the gradients using the chain rule, in the same way as PyTorch did it:

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial c_i} \frac{\partial c_i}{\partial b_i} \frac{\partial b_i}{\partial a_i} \frac{\partial a_i}{\partial x_i}$$

Note that we have simplified this equation to index notation, and by using the fact that all operation besides the mean do not combine the elements in the tensor. The partial derivatives are:

$$\frac{\partial a_i}{\partial x_i} = 1, \quad \frac{\partial b_i}{\partial a_i} = 2 \cdot a_i, \quad \frac{\partial c_i}{\partial b_i} = 1, \quad \frac{\partial y}{\partial c_i} = \frac{1}{3}$$

Hence, with the input being $\mathbf{x} = [0, 1, 2]$, our gradients are $\partial y / \partial \mathbf{x} = [4/3, 2, 8/3]$. The previous code cell should have printed the same result.

GPU support

A crucial feature of PyTorch is the support of GPUs, short for Graphics Processing Unit. A GPU can perform many thousands of small operations in parallel, making it very well suitable for performing large matrix operations in neural networks. When comparing GPUs to CPUs, we can list the following main differences (credit: [Kevin Krewell, 2009](#))

CPU	GPU
Central Processing Unit	Graphics Processing Unit
Several cores	Many cores
Low latency	High throughput
Good for serial processing	Good for parallel processing
Can do a handful of operations at once	Can do thousands of operations at once

CPUs and GPUs have both different advantages and disadvantages, which is why many computers contain both components and use them for different tasks. In case you are not familiar with GPUs, you can read up more details in this [NVIDIA blog post](#) or [here](#).

GPUs can accelerate the training of your network up to a factor of 100 which is essential for large neural networks. PyTorch implements a lot of functionality for supporting GPUs (mostly those of NVIDIA due to the libraries [CUDA](#) and [cuDNN](#)). First, let's check whether you have a GPU available:

[29]: `gpu_avail = torch.cuda.is_available()
print(f"Is the GPU available? {gpu_avail}")`

```
Is the GPU available? True
```

If you have a GPU on your computer but the command above returns False, make sure you have the correct CUDA-version installed. The d12022 environment comes with the CUDA 11.7, which is selected for the Lisa supercomputer. Please change it if necessary (CUDA 11.3 is currently common on Colab). On Google Colab, make sure that you have selected a GPU in your runtime setup (in the menu, check under Runtime -> Change runtime type).

By default, all tensors you create are stored on the CPU. We can push a tensor to the GPU by using the function `.to()`, or `.cuda()`. However, it is often a good practice to define a device object in your code which points to the GPU if you have one, and otherwise to the CPU. Then, you can write your code with respect to this device object, and it allows you to run the same code on both a CPU-only system, and one with a GPU. Let's try it below. We can specify the device as follows:

```
[30]: device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
print("Device", device)
```

Device cuda

Now let's create a tensor and push it to the device:

```
[31]: x = torch.zeros(2, 3)
x = x.to(device)
print("X", x)

X tensor([[0., 0., 0.],
           [0., 0., 0.]], device='cuda:0')
```

In case you have a GPU, you should now see the attribute `device='cuda:0'` being printed next to your tensor. The zero next to `cuda` indicates that this is the zero-th GPU device on your computer. PyTorch also supports multi-GPU systems, but this you will only need once you have very big networks to train (if interested, see the [PyTorch documentation](#)). We can also compare the runtime of a large matrix multiplication on the CPU with a operation on the GPU:

```
[32]: x = torch.randn(5000, 5000)

## CPU version
start_time = time.time()
_ = torch.matmul(x, x)
end_time = time.time()
print(f"CPU time: {(end_time - start_time):.6f}s")

## GPU version
x = x.to(device)
_ = torch.matmul(x, x) # First operation to 'burn in' GPU
# CUDA is asynchronous, so we need to use different timing functions
start = torch.cuda.Event(enable_timing=True)
end = torch.cuda.Event(enable_timing=True)
start.record()
_ = torch.matmul(x, x)
end.record()
torch.cuda.synchronize() # Waits for everything to finish running on the GPU
print(f"GPU time: {0.001 * start.elapsed_time(end):.6f}s") # Milliseconds to seconds

CPU time: 0.20694s
GPU time: 0.00985s
```

Depending on the size of the operation and the CPU/GPU in your system, the speedup of this operation can be $>50\times$. As `matmul` operations are very common in neural networks, we can already see the great benefit of training a NN on a GPU. The time estimate can be relatively noisy here because we haven't run it for multiple times. Feel free to extend this, but it also takes longer to run.

When generating random numbers, the seed between CPU and GPU is not synchronized. Hence, we need to set the seed on the GPU separately to ensure a reproducible code. Note that due to different GPU architectures, running the same code on different GPUs does not guarantee the same random numbers. Still, we don't want that our code gives us a different output every time we run it on the exact same hardware. Hence, we also set the seed on the GPU:

```
[33]: # GPU operations have a separate seed we also want to set
if torch.cuda.is_available():
    torch.cuda.manual_seed(42)
    torch.cuda.manual_seed_all(42)

# Additionally, some operations on a GPU are implemented stochastic for efficiency
# We want to ensure that all operations are deterministic on GPU (if used) for
# reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

4.5.2 Learning by example: Continuous XOR

If we want to build a neural network in PyTorch, we could specify all our parameters (weight matrices, bias vectors) using `Tensors` (with `requires_grad=True`), ask PyTorch to calculate the gradients and then adjust the parameters. But things can quickly get cumbersome if we have a lot of parameters. In PyTorch, there is a package called `torch.nn` that makes building neural networks more convenient.

We will introduce the libraries and all additional parts you might need to train a neural network in PyTorch, using a simple example classifier on a simple yet well known example: XOR. Given two binary inputs x_1 and x_2 , the label to predict is 1 if either x_1 or x_2 is 1 while the other is 0, or the label is 0 in all other cases. The example became famous by the fact that a single neuron, i.e. a linear classifier, cannot learn this simple function. Hence, we will learn how to build a small neural network that can learn this function. To make it a little bit more interesting, we move the XOR into continuous space and introduce some gaussian noise on the binary inputs. Our desired separation of an XOR dataset could look as follows:

The model

The package `torch.nn` defines a series of useful classes like linear networks layers, activation functions, loss functions etc. A full list can be found [here](#). In case you need a certain network layer, check the documentation of the package first before writing the layer yourself as the package likely contains the code for it already. We import it below:

```
[34]: import torch.nn as nn
```

Additionally to `torch.nn`, there is also `torch.nn.functional`. It contains functions that are used in network layers. This is in contrast to `torch.nn` which defines them as `nn.Modules` (more on it below), and `torch.nn` actually uses a lot of functionalities from `torch.nn.functional`. Hence, the functional package is useful in many situations, and so we import it as well here.

```
[35]: import torch.nn.functional as F
```

nn.Module

In PyTorch, a neural network is built up out of modules. Modules can contain other modules, and a neural network is considered to be a module itself as well. The basic template of a module is as follows:

```
[36]: class MyModule(nn.Module):

    def __init__(self):
        super().__init__()
        # Some init for my module

    def forward(self, x):
        # Function for performing the calculation of the module.
        pass
```

The forward function is where the computation of the module is taken place, and is executed when you call the module (`nn = MyModule(); nn(x)`). In the init function, we usually create the parameters of the module, using `nn.Parameter`, or defining other modules that are used in the forward function. The backward calculation is done automatically, but could be overwritten as well if wanted.

Simple classifier

We can now make use of the pre-defined modules in the `torch.nn` package, and define our own small neural network. We will use a minimal network with a input layer, one hidden layer with `tanh` as activation function, and a output layer. In other words, our networks should look something like this:

The input neurons are shown in blue, which represent the coordinates x_1 and x_2 of a data point. The hidden neurons including a `tanh` activation are shown in white, and the output neuron in red. In PyTorch, we can define this as follows:

```
[37]: class SimpleClassifier(nn.Module):

    def __init__(self, num_inputs, num_hidden, num_outputs):
        super().__init__()
        # Initialize the modules we need to build the network
        self.linear1 = nn.Linear(num_inputs, num_hidden)
        self.act_fn = nn.Tanh()
        self.linear2 = nn.Linear(num_hidden, num_outputs)

    def forward(self, x):
        # Perform the calculation of the model to determine the prediction
        x = self.linear1(x)
        x = self.act_fn(x)
        x = self.linear2(x)
        return x
```

For the examples in this notebook, we will use a tiny neural network with two input neurons and four hidden neurons. As we perform binary classification, we will use a single output neuron. Note that we do not apply a sigmoid on the output yet. This is because other functions, especially the loss, are more efficient and precise to calculate on the original outputs instead of the sigmoid output. We will discuss the detailed reason later.

```
[38]: model = SimpleClassifier(num_inputs=2, num_hidden=4, num_outputs=1)
# Printing a module shows all its submodules
print(model)

SimpleClassifier(
    (linear1): Linear(in_features=2, out_features=4, bias=True)
    (act_fn): Tanh()
    (linear2): Linear(in_features=4, out_features=1, bias=True)
)
```

Printing the model lists all submodules it contains. The parameters of a module can be obtained by using its `parameters()` functions, or `named_parameters()` to get a name to each parameter object. For our small neural network, we have the following parameters:

```
[39]: for name, param in model.named_parameters():
    print(f"Parameter {name}, shape {param.shape}")

Parameter linear1.weight, shape torch.Size([4, 2])
Parameter linear1.bias, shape torch.Size([4])
Parameter linear2.weight, shape torch.Size([1, 4])
Parameter linear2.bias, shape torch.Size([1])
```

Each linear layer has a weight matrix of the shape [output, input], and a bias of the shape [output]. The tanh activation function does not have any parameters. Note that parameters are only registered for `nn.Module` objects that are direct object attributes, i.e. `self.a = ...`. If you define a list of modules, the parameters of those are not registered for the outer module and can cause some issues when you try to optimize your module. There are alternatives, like `nn.ModuleList`, `nn.ModuleDict` and `nn.Sequential`, that allow you to have different data structures of modules. We will use them in a few later tutorials and explain them there.

The data

PyTorch also provides a few functionalities to load the training and test data efficiently, summarized in the package `torch.utils.data`.

```
[40]: import torch.utils.data as data
```

The data package defines two classes which are the standard interface for handling data in PyTorch: `data.Dataset`, and `data.DataLoader`. The dataset class provides an uniform interface to access the training/test data, while the data loader makes sure to efficiently load and stack the data points from the dataset into batches during training.

The dataset class

The dataset class summarizes the basic functionality of a dataset in a natural way. To define a dataset in PyTorch, we simply specify two functions: `__getitem__`, and `__len__`. The get-item function has to return the *i*-th data point in the dataset, while the len function returns the size of the dataset. For the XOR dataset, we can define the dataset class as follows:

```
[41]: class XORDataset(data.Dataset):

    def __init__(self, size, std=0.1):
        """
        Inputs:
```

(continues on next page)

(continued from previous page)

```

size - Number of data points we want to generate
std - Standard deviation of the noise (see generate_continuous_xor function)
"""

super().__init__()
self.size = size
self.std = std
self.generate_continuous_xor()

def generate_continuous_xor(self):
    # Each data point in the XOR dataset has two variables, x and y, that can be either 0 or 1
    # The label is their XOR combination, i.e. 1 if only x or only y is 1 while the other is 0.
    # If x=y, the label is 0.
    data = torch.randint(low=0, high=2, size=(self.size, 2), dtype=torch.float32)
    label = (data.sum(dim=1) == 1).to(torch.long)
    # To make it slightly more challenging, we add a bit of gaussian noise to the data points.
    data += self.std * torch.randn(data.shape)

    self.data = data
    self.label = label

def __len__(self):
    # Number of data point we have. Alternatively self.data.shape[0], or self.label.shape[0]
    return self.size

def __getitem__(self, idx):
    # Return the idx-th data point of the dataset
    # If we have multiple things to return (data point and label), we can return them as tuple
    data_point = self.data[idx]
    data_label = self.label[idx]
    return data_point, data_label

```

Let's try to create such a dataset and inspect it:

```
[42]: dataset = XORDataset(size=200)
print("Size of dataset:", len(dataset))
print("Data point 0:", dataset[0])

Size of dataset: 200
Data point 0: (tensor([0.9632, 0.1117]), tensor(1))
```

To better relate to the dataset, we visualize the samples below.

```
[43]: def visualize_samples(data, label):
    if isinstance(data, torch.Tensor):
        data = data.cpu().numpy()
    if isinstance(label, torch.Tensor):
        label = label.cpu().numpy()
    data_0 = data[label == 0]
```

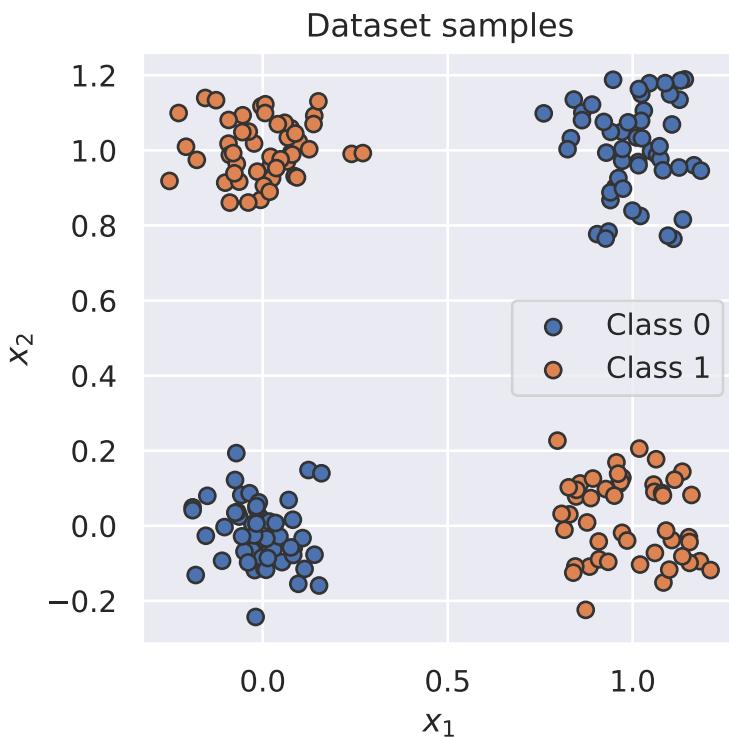
(continues on next page)

(continued from previous page)

```
data_1 = data[label == 1]

plt.figure(figsize=(4,4))
plt.scatter(data_0[:,0], data_0[:,1], edgecolor="#333", label="Class 0")
plt.scatter(data_1[:,0], data_1[:,1], edgecolor="#333", label="Class 1")
plt.title("Dataset samples")
plt.ylabel(r"$x_2$")
plt.xlabel(r"$x_1$")
plt.legend()
```

[44]: `visualize_samples(dataset.data, dataset.label)`
`plt.show()`



The data loader class

The class `torch.utils.data.DataLoader` represents a Python iterable over a dataset with support for automatic batching, multi-process data loading and many more features. The data loader communicates with the dataset using the function `__getitem__`, and stacks its outputs as tensors over the first dimension to form a batch. In contrast to the dataset class, we usually don't have to define our own data loader class, but can just create an object of it with the dataset as input. Additionally, we can configure our data loader with the following input arguments (only a selection, see full list [here](#)):

- `batch_size`: Number of samples to stack per batch
- `shuffle`: If True, the data is returned in a random order. This is important during training for introducing stochasticity.
- `num_workers`: Number of subprocesses to use for data loading. The default, 0, means that the data will be loaded

in the main process which can slow down training for datasets where loading a data point takes a considerable amount of time (e.g. large images). More workers are recommended for those, but can cause issues on Windows computers. For tiny datasets as ours, 0 workers are usually faster.

- `pin_memory`: If True, the data loader will copy Tensors into CUDA pinned memory before returning them. This can save some time for large data points on GPUs. Usually a good practice to use for a training set, but not necessarily for validation and test to save memory on the GPU.
- `drop_last`: If True, the last batch is dropped in case it is smaller than the specified batch size. This occurs when the dataset size is not a multiple of the batch size. Only potentially helpful during training to keep a consistent batch size.

Let's create a simple data loader below:

```
[45]: data_loader = data.DataLoader(dataset, batch_size=8, shuffle=True)
```

```
[46]: # next(iter(...)) catches the first batch of the data loader
# If shuffle is True, this will return a different batch every time we run this cell
# For iterating over the whole dataset, we can simply use "for batch in data_loader: ..."
data_inputs, data_labels = next(iter(data_loader))

# The shape of the outputs are [batch_size, d_1,...,d_N] where d_1,...,d_N are the
# dimensions of the data point returned from the dataset class
print("Data inputs", data_inputs.shape, "\n", data_inputs)
print("Data labels", data_labels.shape, "\n", data_labels)

Data inputs torch.Size([8, 2])
tensor([[-0.0890,  0.8608],
       [ 1.0905, -0.0128],
       [ 0.7967,  0.2268],
       [-0.0688,  0.0371],
       [ 0.8732, -0.2240],
       [-0.0559, -0.0282],
       [ 0.9277,  0.0978],
       [ 1.0150,  0.9689]])
Data labels torch.Size([8])
tensor([1, 1, 1, 0, 1, 0, 1, 0])
```

Optimization

After defining the model and the dataset, it is time to prepare the optimization of the model. During training, we will perform the following steps:

1. Get a batch from the data loader
2. Obtain the predictions from the model for the batch
3. Calculate the loss based on the difference between predictions and labels
4. Backpropagation: calculate the gradients for every parameter with respect to the loss
5. Update the parameters of the model in the direction of the gradients

We have seen how we can do step 1, 2 and 4 in PyTorch. Now, we will look at step 3 and 5.

Loss modules

We can calculate the loss for a batch by simply performing a few tensor operations as those are automatically added to the computation graph. For instance, for binary classification, we can use Binary Cross Entropy (BCE) which is defined as follows:

$$\mathcal{L}_{BCE} = - \sum_i [y_i \log x_i + (1 - y_i) \log(1 - x_i)]$$

where y are our labels, and x our predictions, both in the range $[0, 1]$. However, PyTorch already provides a list of predefined loss functions which we can use (see [here](#) for a full list). For instance, for BCE, PyTorch has two modules: `nn.BCELoss()`, `nn.BCEWithLogitsLoss()`. While `nn.BCELoss` expects the inputs x to be in the range $[0, 1]$, i.e. the output of a sigmoid, `nn.BCEWithLogitsLoss` combines a sigmoid layer and the BCE loss in a single class. This version is numerically more stable than using a plain Sigmoid followed by a BCE loss because of the logarithms applied in the loss function. Hence, it is advised to use loss functions applied on “logits” where possible (remember to not apply a sigmoid on the output of the model in this case!). For our model defined above, we therefore use the module `nn.BCEWithLogitsLoss`.

```
[47]: loss_module = nn.BCEWithLogitsLoss()
```

Stochastic Gradient Descent

For updating the parameters, PyTorch provides the package `torch.optim` that has most popular optimizers implemented. We will discuss the specific optimizers and their differences later in the course, but will for now use the simplest of them: `torch.optim.SGD`. Stochastic Gradient Descent updates parameters by multiplying the gradients with a small constant, called learning rate, and subtracting those from the parameters (hence minimizing the loss). Therefore, we slowly move towards the direction of minimizing the loss. A good default value of the learning rate for a small network as ours is 0.1.

```
[48]: # Input to the optimizer are the parameters of the model: model.parameters()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

The optimizer provides two useful functions: `optimizer.step()`, and `optimizer.zero_grad()`. The step function updates the parameters based on the gradients as explained above. The function `optimizer.zero_grad()` sets the gradients of all parameters to zero. While this function seems less relevant at first, it is a crucial pre-step before performing backpropagation. If we call the `backward` function on the loss while the parameter gradients are non-zero from the previous batch, the new gradients would actually be added to the previous ones instead of overwriting them. This is done because a parameter might occur multiple times in a computation graph, and we need to sum the gradients in this case instead of replacing them. Hence, remember to call `optimizer.zero_grad()` before calculating the gradients of a batch.

Training

Finally, we are ready to train our model. As a first step, we create a slightly larger dataset and specify a data loader with a larger batch size.

```
[49]: train_dataset = XORDataset(size=2500)
train_data_loader = data.DataLoader(train_dataset, batch_size=128, shuffle=True)
```

Now, we can write a small training function. Remember our five steps: load a batch, obtain the predictions, calculate the loss, backpropagate, and update. Additionally, we have to push all data and model parameters to the device of our choice (GPU if available). For the tiny neural network we have, communicating the data to the GPU actually takes much more time than we could save from running the operation on GPU. For large networks, the communication time

is significantly smaller than the actual runtime making a GPU crucial in these cases. Still, to practice, we will push the data to GPU here.

```
[50]: # Push model to device. Has to be only done once
model.to(device)

[50]: SimpleClassifier(
    (linear1): Linear(in_features=2, out_features=4, bias=True)
    (act_fn): Tanh()
    (linear2): Linear(in_features=4, out_features=1, bias=True)
)
```

In addition, we set our model to training mode. This is done by calling `model.train()`. There exist certain modules that need to perform a different forward step during training than during testing (e.g. BatchNorm and Dropout), and we can switch between them using `model.train()` and `model.eval()`.

```
[51]: def train_model(model, optimizer, data_loader, loss_module, num_epochs=100):
    # Set model to train mode
    model.train()

    # Training loop
    for epoch in tqdm(range(num_epochs)):
        for data_inputs, data_labels in data_loader:

            ## Step 1: Move input data to device (only strictly necessary if we use GPU)
            data_inputs = data_inputs.to(device)
            data_labels = data_labels.to(device)

            ## Step 2: Run the model on the input data
            preds = model(data_inputs)
            preds = preds.squeeze(dim=1) # Output is [Batch size, 1], but we want [Batch_size]

            ## Step 3: Calculate the loss
            loss = loss_module(preds, data_labels.float())

            ## Step 4: Perform backpropagation
            # Before calculating the gradients, we need to ensure that they are all zero.
            # The gradients would not be overwritten, but actually added to the existing_ones.
            optimizer.zero_grad()
            # Perform backpropagation
            loss.backward()

            ## Step 5: Update the parameters
            optimizer.step()
```

```
[52]: train_model(model, optimizer, train_data_loader, loss_module)
0% | 0/100 [00:00<?, ?it/s]
```

Saving a model

After finish training a model, we save the model to disk so that we can load the same weights at a later time. For this, we extract the so-called `state_dict` from the model which contains all learnable parameters. For our simple model, the state dict contains the following entries:

```
[53]: state_dict = model.state_dict()
print(state_dict)

OrderedDict([('linear1.weight', tensor([-2.6034, -3.3292],
    [ 1.9774, -2.4076],
    [-2.5968, -1.5908],
    [-0.5717, -0.8101]), device='cuda:0'), ('linear1.bias', tensor([ 1.4459, -1.
    3992,  2.9882, -0.1375], device='cuda:0'), ('linear2.weight', tensor([-4.4623,  3.
    0885,  4.4030, -0.1377]), device='cuda:0'), ('linear2.bias', tensor([-1.6853], device=
    'cuda:0'))])
```

To save the state dictionary, we can use `torch.save`:

```
[54]: # torch.save(object, filename). For the filename, any extension can be used
torch.save(state_dict, "our_model.tar")
```

To load a model from a state dict, we use the function `torch.load` to load the state dict from the disk, and the module function `load_state_dict` to overwrite our parameters with the new values:

```
[55]: # Load state dict from the disk (make sure it is the same name as above)
state_dict = torch.load("our_model.tar")

# Create a new model and load the state
new_model = SimpleClassifier(num_inputs=2, num_hidden=4, num_outputs=1)
new_model.load_state_dict(state_dict)

# Verify that the parameters are the same
print("Original model\n", model.state_dict())
print("\nLoaded model\n", new_model.state_dict())

Original model
OrderedDict([('linear1.weight', tensor([-2.6034, -3.3292],
    [ 1.9774, -2.4076],
    [-2.5968, -1.5908],
    [-0.5717, -0.8101]), device='cuda:0'), ('linear1.bias', tensor([ 1.4459, -1.
    3992,  2.9882, -0.1375], device='cuda:0'), ('linear2.weight', tensor([-4.4623,  3.
    0885,  4.4030, -0.1377]), device='cuda:0'), ('linear2.bias', tensor([-1.6853], device=
    'cuda:0'))])

Loaded model
OrderedDict([('linear1.weight', tensor([-2.6034, -3.3292],
    [ 1.9774, -2.4076],
    [-2.5968, -1.5908],
    [-0.5717, -0.8101])), ('linear1.bias', tensor([ 1.4459, -1.3992,  2.9882, -0.
    1375])), ('linear2.weight', tensor([-4.4623,  3.0885,  4.4030, -0.1377])), ('linear2.
    bias', tensor([-1.6853]))])
```

A detailed tutorial on saving and loading models in PyTorch can be found [here](#).

Evaluation

Once we have trained a model, it is time to evaluate it on a held-out test set. As our dataset consist of randomly generated data points, we need to first create a test set with a corresponding data loader.

```
[56]: test_dataset = XORDataset(size=500)
# drop_last -> Don't drop the last batch although it is smaller than 128
test_data_loader = data.DataLoader(test_dataset, batch_size=128, shuffle=False, drop_
↪last=False)
```

As metric, we will use accuracy which is calculated as follows:

$$acc = \frac{\text{#correct predictions}}{\text{\#all predictions}} = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP are the true positives, TN true negatives, FP false positives, and FN the false negatives.

When evaluating the model, we don't need to keep track of the computation graph as we don't intend to calculate the gradients. This reduces the required memory and speed up the model. In PyTorch, we can deactivate the computation graph using `with torch.no_grad():` Remember to additionally set the model to eval mode.

```
[57]: def eval_model(model, data_loader):
    model.eval() # Set model to eval mode
    true_preds, num_preds = 0., 0.

    with torch.no_grad(): # Deactivate gradients for the following code
        for data_inputs, data_labels in data_loader:

            # Determine prediction of model on dev set
            data_inputs, data_labels = data_inputs.to(device), data_labels.to(device)
            preds = model(data_inputs)
            preds = preds.squeeze(dim=1)
            preds = torch.sigmoid(preds) # Sigmoid to map predictions between 0 and 1
            pred_labels = (preds >= 0.5).long() # Binarize predictions to 0 and 1

            # Keep records of predictions for the accuracy metric (true_preds=TP+TN, num_
            ↪preds=TP+TN+FP+FN)
            true_preds += (pred_labels == data_labels).sum()
            num_preds += data_labels.shape[0]

    acc = true_preds / num_preds
    print(f"Accuracy of the model: {100.*acc:.2f}%")
```

```
[58]: eval_model(model, test_data_loader)
```

```
Accuracy of the model: 100.00%
```

If we trained our model correctly, we should see a score close to 100% accuracy. However, this is only possible because of our simple task, and unfortunately, we usually don't get such high scores on test sets of more complex tasks.

Visualizing classification boundaries

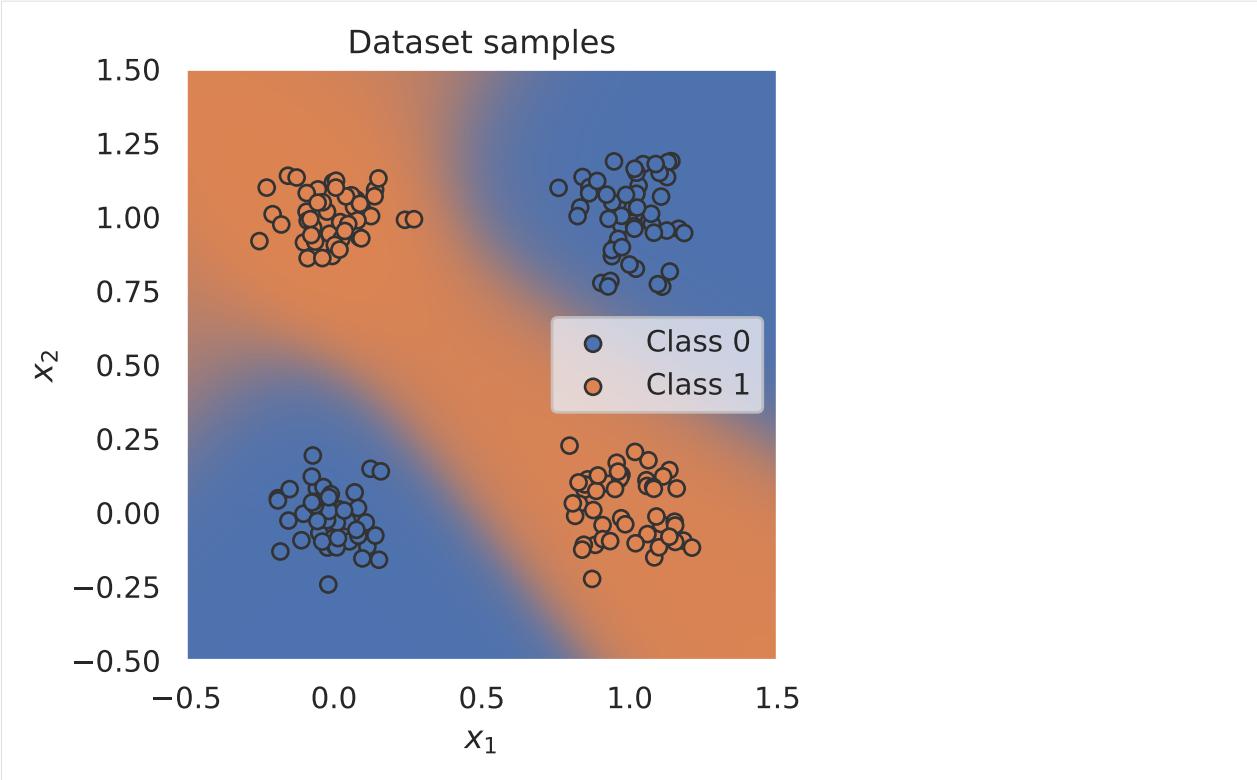
To visualize what our model has learned, we can perform a prediction for every data point in a range of $[-0.5, 1.5]$, and visualize the predicted class as in the sample figure at the beginning of this section. This shows where the model has created decision boundaries, and which points would be classified as 0, and which as 1. We therefore get a background image out of blue (class 0) and orange (class 1). The spots where the model is uncertain we will see a blurry overlap. The specific code is less relevant compared to the output figure which should hopefully show us a clear separation of classes:

```
[59]: @torch.no_grad() # Decorator, same effect as "with torch.no_grad(): ..." over the whole function.
def visualize_classification(model, data, label):
    if isinstance(data, torch.Tensor):
        data = data.cpu().numpy()
    if isinstance(label, torch.Tensor):
        label = label.cpu().numpy()
    data_0 = data[label == 0]
    data_1 = data[label == 1]

    fig = plt.figure(figsize=(4,4), dpi=500)
    plt.scatter(data_0[:,0], data_0[:,1], edgecolor="#333", label="Class 0")
    plt.scatter(data_1[:,0], data_1[:,1], edgecolor="#333", label="Class 1")
    plt.title("Dataset samples")
    plt.ylabel(r"$x_2$")
    plt.xlabel(r"$x_1$")
    plt.legend()

    # Let's make use of a lot of operations we have learned above
    model.to(device)
    c0 = torch.Tensor(to_rgba("C0")).to(device)
    c1 = torch.Tensor(to_rgba("C1")).to(device)
    x1 = torch.arange(-0.5, 1.5, step=0.01, device=device)
    x2 = torch.arange(-0.5, 1.5, step=0.01, device=device)
    xx1, xx2 = torch.meshgrid(x1, x2, indexing='ij') # Meshgrid function as in numpy
    model_inputs = torch.stack([xx1, xx2], dim=-1)
    preds = model(model_inputs)
    preds = torch.sigmoid(preds)
    output_image = (1 - preds) * c0[None,None] + preds * c1[None,None] # Specifying
    # "None" in a dimension creates a new one
    output_image = output_image.cpu().numpy() # Convert to numpy array. This only works
    # for tensors on CPU, hence first push to CPU
    plt.imshow(output_image, origin='lower', extent=(-0.5, 1.5, -0.5, 1.5))
    plt.grid(False)
    return fig

_ = visualize_classification(model, dataset.data, dataset.label)
plt.show()
```



The decision boundaries might not look exactly as in the figure in the preamble of this section which can be caused by running it on CPU or a different GPU architecture. Nevertheless, the result on the accuracy metric should be the approximately the same.

4.5.3 Additional features we didn't get to discuss yet

Finally, you are all set to start with your own PyTorch project! In summary, we have looked at how we can build neural networks in PyTorch, and train and test them on data. However, there is still much more to PyTorch we haven't discussed yet. In the coming series of Jupyter notebooks, we will discover more and more functionalities of PyTorch, so that you also get familiar to PyTorch concepts beyond the basics. If you are already interested in learning more of PyTorch, we recommend the official [tutorial website](#) that contains many tutorials on various topics. Especially logging with Tensorboard ([official tutorial here](#)) is a good practice that we will explore further from Tutorial 5 on in combination with PyTorch Lightning. Nonetheless, let's check it shortly out how we could use TensorBoard in our small example.

TensorBoard logging

TensorBoard is a logging and visualization tool that is a popular choice for training deep learning models. Although initially published for TensorFlow, TensorBoard is also integrated in PyTorch allowing us to easily use it. First, let's import it below.

```
[60]: # Import tensorboard logger from PyTorch
from torch.utils.tensorboard import SummaryWriter

# Load tensorboard extension for Jupyter Notebook, only need to start TB in the notebook
%load_ext tensorboard
```

The last line is required if you want to run TensorBoard directly in the Jupyter Notebook. Otherwise, you can start TensorBoard from the terminal.

PyTorch's TensorBoard API is simple to use. We start the logging process by creating a new object, `writer = SummaryWriter(...)`, where we specify the directory in which the logging file should be saved. With this object, we can log different aspects of our model by calling functions of the style `writer.add_....`. For example, we can visualize the computation graph with the function `writer.add_graph`, or add a scalar value like the loss with `writer.add_scalar`. Let's adapt our initial training function with adding a TensorBoard logger below.

```
[61]: def train_model_with_logger(model, optimizer, data_loader, loss_module, val_dataset, num_
→epochs=100, logging_dir='runs/our_experiment'):

    # Create TensorBoard logger
    writer = SummaryWriter(logging_dir)
    model_plotted = False

    # Set model to train mode
    model.train()

    # Training loop
    for epoch in tqdm(range(num_epochs)):
        epoch_loss = 0.0
        for data_inputs, data_labels in data_loader:

            ## Step 1: Move input data to device (only strictly necessary if we use GPU)
            data_inputs = data_inputs.to(device)
            data_labels = data_labels.to(device)

            # For the very first batch, we visualize the computation graph in TensorBoard
            if not model_plotted:
                writer.add_graph(model, data_inputs)
                model_plotted = True

            ## Step 2: Run the model on the input data
            preds = model(data_inputs)
            preds = preds.squeeze(dim=1) # Output is [Batch size, 1], but we want [Batch_
→size]

            ## Step 3: Calculate the loss
            loss = loss_module(preds, data_labels.float())

            ## Step 4: Perform backpropagation
            # Before calculating the gradients, we need to ensure that they are all zero.
            # The gradients would not be overwritten, but actually added to the existing_
→ones.
            optimizer.zero_grad()
            # Perform backpropagation
            loss.backward()

            ## Step 5: Update the parameters
            optimizer.step()

            ## Step 6: Take the running average of the loss
            epoch_loss += loss.item()
```

(continues on next page)

(continued from previous page)

```
# Add average loss to TensorBoard
epoch_loss /= len(data_loader)
writer.add_scalar('training_loss',
                  epoch_loss,
                  global_step = epoch + 1)

# Visualize prediction and add figure to TensorBoard
# Since matplotlib figures can be slow in rendering, we only do it every 10th
↪epoch
if (epoch + 1) % 10 == 0:
    fig = visualize_classification(model, val_dataset.data, val_dataset.label)
    writer.add_figure('predictions',
                      fig,
                      global_step = epoch + 1)

writer.close()
```

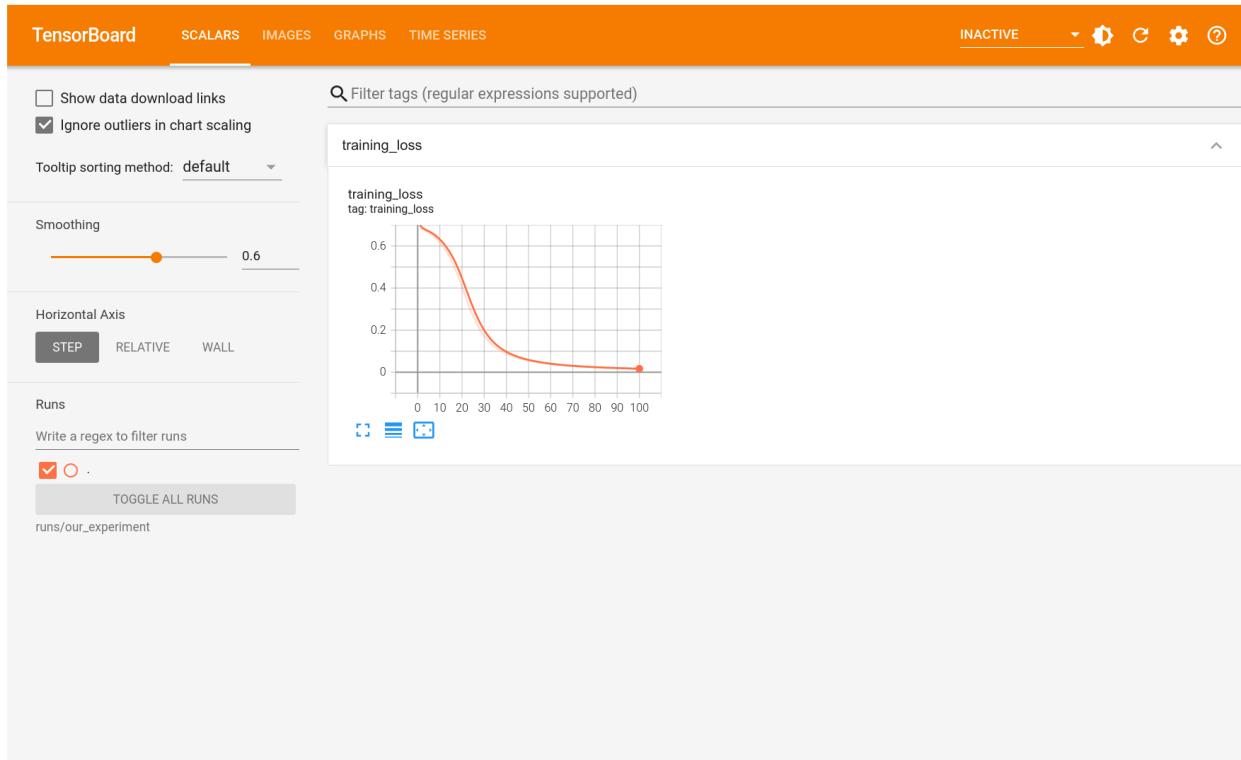
Let's use this method to train a model as before, with a new model and optimizer.

```
[62]: model = SimpleClassifier(num_inputs=2, num_hidden=4, num_outputs=1).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
train_model_with_logger(model, optimizer, train_data_loader, loss_module, val_
↪dataset=dataset)

0% | 0/100 [00:00<?, ?it/s]
```

The TensorBoard file in the folder `runs/our_experiment` now contains a loss curve, the computation graph of our network, and a visualization of the learned predictions over number of epochs. To start the TensorBoard visualizer, simply run the following statement:

```
[63]: %tensorboard --logdir runs/our_experiment
```



TensorBoard visualizations can help to identify possible issues with your model, and identify situations such as overfitting. You can also track the training progress while a model is training, since the logger automatically writes everything added to it to the logging file. Feel free to explore the TensorBoard functionalities, and we will make use of TensorBoards a couple of times from Tutorial 5 on.

If you found this tutorial helpful, consider -ing our repository.
 For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.6 Tutorial 3: Activation Functions

Filled notebook:

Pre-trained models:

Recordings:

JAX+Flax version:

Author: Phillip Lippe

Note: Interested in JAX? Check out our [JAX+Flax version](#) of this tutorial!

In this tutorial, we will take a closer look at (popular) activation functions and investigate their effect on optimization properties in neural networks. Activation functions are a crucial part of deep learning models as they add the non-linearity to neural networks. There is a great variety of activation functions in the literature, and some are more beneficial than others. The goal of this tutorial is to show the importance of choosing a good activation function (and how to do so), and what problems might occur if we don't.

Before we start, we import our standard libraries and set up basic functions:

```
[1]: ## Standard libraries
import os
import json
import math
import numpy as np

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
import seaborn as sns
sns.set()

## Progress bar
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
```

We will define a function to set a seed on all libraries we might interact with in this tutorial (here numpy and torch). This allows us to make our training reproducible. However, note that in contrast to the CPU, the same seed on different GPU architectures can give different results. All models here have been trained on an NVIDIA GTX1080Ti, which is also the GPU type provided by the Lisa gpu_shared_course partition.

Additionally, the following cell defines two paths: DATASET_PATH and CHECKPOINT_PATH. The dataset path is the directory where we will download datasets used in the notebooks. It is recommended to store all datasets from PyTorch in one joined directory to prevent duplicate downloads. The checkpoint path is the directory where we will store trained model weights and additional files. The needed files will be automatically downloaded. In case you are on Google Colab, it is recommended to change the directories to start from the current directory (i.e. remove `../` for both dataset and checkpoint path).

```
[2]: # Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial3"

# Function for setting the seed
def set_seed(seed):
```

(continues on next page)

(continued from previous page)

```

np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available(): # GPU operation have separate seed
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
set_seed(42)

# Additionally, some operations on a GPU are implemented stochastic for efficiency
# We want to ensure that all operations are deterministic on GPU (if used) for
# reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Fetching the device that will be used throughout this notebook
device = torch.device("cpu") if not torch.cuda.is_available() else torch.device("cuda:0")
print("Using device", device)

Using device cuda:0

```

The following cell downloads all pretrained models we will use in this notebook. The files are stored on a separate repository to reduce the size of the notebook repository, especially for building the documentation on ReadTheDocs. In case the download below fails, you can download the models from a [Google Drive folder](#). Please let me (Phillip) know if an error occurs so it can be fixed for all students.

```

[3]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial3/"
# Files to download
pretrained_files = ["FashionMNIST_elu.config", "FashionMNIST_elu.tar",
                    "FashionMNIST_leakyrelu.config", "FashionMNIST_leakyrelu.tar",
                    "FashionMNIST_relu.config", "FashionMNIST_relu.tar",
                    "FashionMNIST_sigmoid.config", "FashionMNIST_sigmoid.tar",
                    "FashionMNIST_swish.config", "FashionMNIST_swish.tar",
                    "FashionMNIST_tanh.config", "FashionMNIST_tanh.tar"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive"
                  "folder, or contact the author with the full output including the following error:\n", e)

```

4.6.1 Common activation functions

As a first step, we will implement some common activation functions by ourselves. Of course, most of them can also be found in the `torch.nn` package (see the [documentation](#) for an overview). However, we'll write our own functions here for a better understanding and insights.

For an easier time of comparing various activation functions, we start with defining a base class from which all our future modules will inherit:

```
[4]: class ActivationFunction(nn.Module):

    def __init__(self):
        super().__init__()
        self.name = self.__class__.__name__
        self.config = {"name": self.name}
```

Every activation function will be an `nn.Module` so that we can integrate them nicely in a network. We will use the `config` dictionary to store adjustable parameters for some activation functions.

Next, we implement two of the “oldest” activation functions that are still commonly used for various tasks: sigmoid and tanh. Both the sigmoid and tanh activation can be also found as PyTorch functions (`torch.sigmoid`, `torch.tanh`) or as modules (`nn.Sigmoid`, `nn.Tanh`). Here, we implement them by hand:

```
[5]: #####
class Sigmoid(ActivationFunction):

    def forward(self, x):
        return 1 / (1 + torch.exp(-x))

#####
class Tanh(ActivationFunction):

    def forward(self, x):
        x_exp, neg_x_exp = torch.exp(x), torch.exp(-x)
        return (x_exp - neg_x_exp) / (x_exp + neg_x_exp)

#####
```

Another popular activation function that has allowed the training of deeper networks, is the Rectified Linear Unit (ReLU). Despite its simplicity of being a piecewise linear function, ReLU has one major benefit compared to sigmoid and tanh: a strong, stable gradient for a large range of values. Based on this idea, a lot of variations of ReLU have been proposed, of which we will implement the following three: LeakyReLU, ELU, and Swish. LeakyReLU replaces the zero settings in the negative part with a smaller slope to allow gradients to flow also in this part of the input. Similarly, ELU replaces the negative part with an exponential decay. The third, most recently proposed activation function is Swish, which is actually the result of a large experiment with the purpose of finding the “optimal” activation function. Compared to the other activation functions, Swish is both smooth and non-monotonic (i.e. contains a change of sign in the gradient). This has been shown to prevent dead neurons as in standard ReLU activation, especially for deep networks. If interested, a more detailed discussion of the benefits of Swish can be found in [this paper](#) [1].

Let's implement the four activation functions below:

```
[6]: #####
```

(continues on next page)

(continued from previous page)

```

class ReLU(ActivationFunction):

    def forward(self, x):
        return x * (x > 0).float()

#####
class LeakyReLU(ActivationFunction):

    def __init__ (self, alpha=0.1):
        super().__init__()
        self.config["alpha"] = alpha

    def forward(self, x):
        return torch.where(x > 0, x, self.config["alpha"] * x)

#####
class ELU(ActivationFunction):

    def forward(self, x):
        return torch.where(x > 0, x, torch.exp(x)-1)

#####
class Swish(ActivationFunction):

    def forward(self, x):
        return x * torch.sigmoid(x)

#####

```

For later usage, we summarize all our activation functions in a dictionary mapping the name to the class object. In case you implement a new activation function by yourself, add it here to include it in future comparisons as well:

```
[7]: act_fn_by_name = {
    "sigmoid": Sigmoid,
    "tanh": Tanh,
    "relu": ReLU,
    "leakyrelu": LeakyReLU,
    "elu": ELU,
    "swish": Swish
}
```

Visualizing activation functions

To get an idea of what each activation function actually does, we will visualize them in the following. Next to the actual activation value, the gradient of the function is an important aspect as it is crucial for optimizing the neural network. PyTorch allows us to compute the gradients simply by calling the `backward` function:

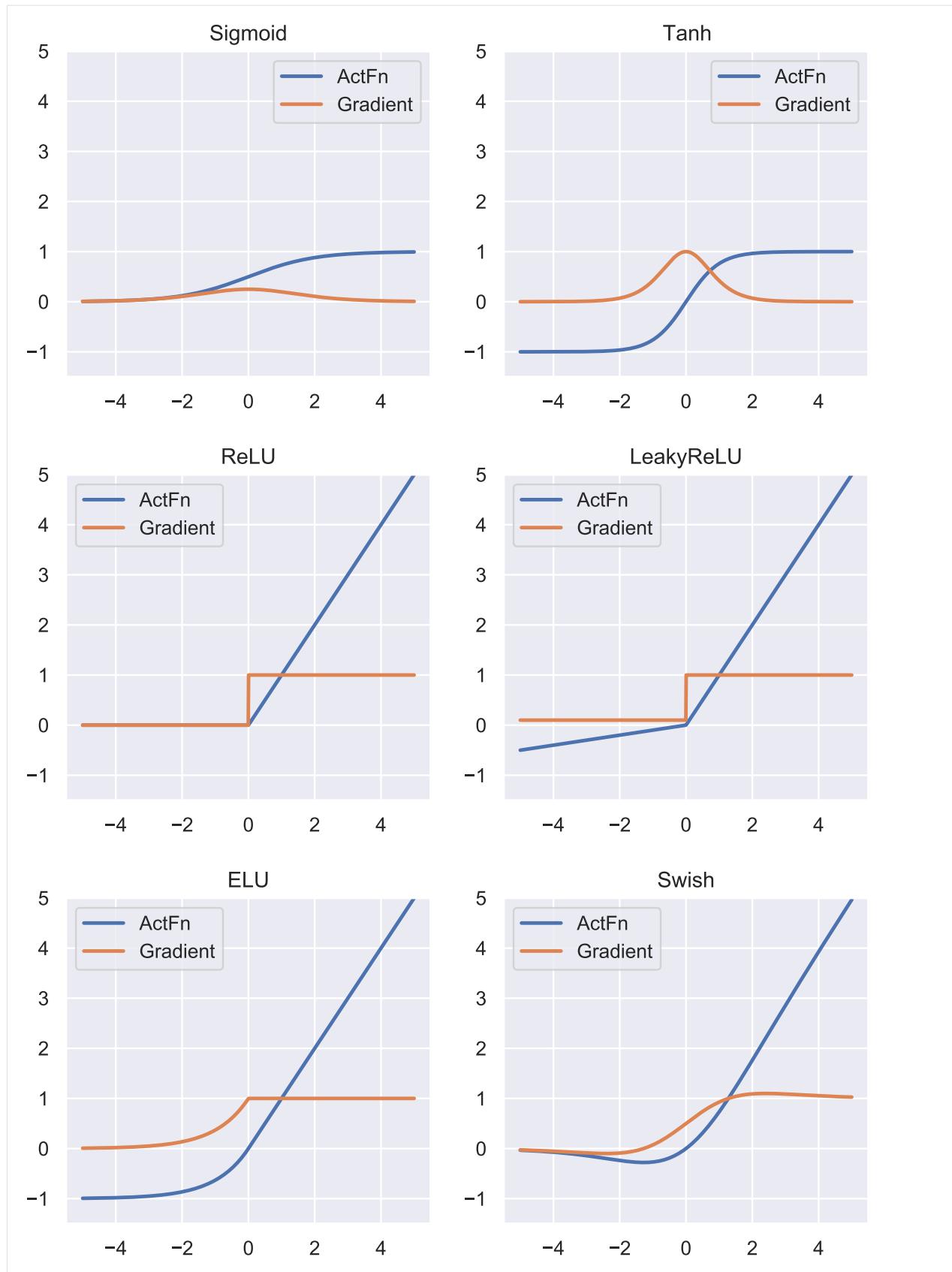
```
[8]: def get_grads(act_fn, x):
    """
    Computes the gradients of an activation function at specified positions.

    Inputs:
        act_fn - An object of the class "ActivationFunction" with an implemented forward_
    ↪ pass.
        x - 1D input tensor.
    Output:
        A tensor with the same size of x containing the gradients of act_fn at x.
    """
    x = x.clone().requires_grad_() # Mark the input as tensor for which we want to store_
    ↪ gradients
    out = act_fn(x)
    out.sum().backward() # Summing results in an equal gradient flow to each element in x
    return x.grad # Accessing the gradients of x by "x.grad"
```

Now we can visualize all our activation functions including their gradients:

```
[9]: def vis_act_fn(act_fn, ax, x):
    # Run activation function
    y = act_fn(x)
    y_grads = get_grads(act_fn, x)
    # Push x, y and gradients back to cpu for plotting
    x, y, y_grads = x.cpu().numpy(), y.cpu().numpy(), y_grads.cpu().numpy()
    ## Plotting
    ax.plot(x, y, linewidth=2, label="ActFn")
    ax.plot(x, y_grads, linewidth=2, label="Gradient")
    ax.set_title(act_fn.name)
    ax.legend()
    ax.set_ylim(-1.5, x.max())

    # Add activation functions if wanted
    act_fns = [act_fn() for act_fn in act_fn_by_name.values()]
    x = torch.linspace(-5, 5, 1000) # Range on which we want to visualize the activation_
    ↪ functions
    ## Plotting
    rows = math.ceil(len(act_fns)/2.0)
    fig, ax = plt.subplots(rows, 2, figsize=(8, rows*4))
    for i, act_fn in enumerate(act_fns):
        vis_act_fn(act_fn, ax[divmod(i, 2)], x)
    fig.subplots_adjust(hspace=0.3)
    plt.show()
```



4.6.2 Analysing the effect of activation functions

After implementing and visualizing the activation functions, we are aiming to gain insights into their effect. We do this by using a simple neural network trained on [FashionMNIST](#) and examine various aspects of the model, including the performance and gradient flow.

Setup

Firstly, let's set up a neural network. The chosen network views the images as 1D tensors and pushes them through a sequence of linear layers and a specified activation function. Feel free to experiment with other network architectures.

```
[10]: class BaseNetwork(nn.Module):

    def __init__(self, act_fn, input_size=784, num_classes=10, hidden_sizes=[512, 256, ↵256, 128]):
        """
        Inputs:
            act_fn - Object of the activation function that should be used as non-linearity in the network.
            input_size - Size of the input images in pixels
            num_classes - Number of classes we want to predict
            hidden_sizes - A list of integers specifying the hidden layer sizes in the NN
        """
        super().__init__()

        # Create the network based on the specified hidden sizes
        layers = []
        layer_sizes = [input_size] + hidden_sizes
        for layer_index in range(1, len(layer_sizes)):
            layers += [nn.Linear(layer_sizes[layer_index-1], layer_sizes[layer_index]), act_fn]
        layers += [nn.Linear(layer_sizes[-1], num_classes)]
        self.layers = nn.Sequential(*layers) # nn.Sequential summarizes a list of modules into a single module, applying them in sequence

        # We store all hyperparameters in a dictionary for saving and loading of the model
        self.config = {"act_fn": act_fn.config, "input_size": input_size, "num_classes": num_classes, "hidden_sizes": hidden_sizes}

    def forward(self, x):
        x = x.view(x.size(0), -1) # Reshape images to a flat vector
        out = self.layers(x)
        return out
```

We also add functions for loading and saving the model. The hyperparameters are stored in a configuration file (simple json file):

```
[11]: def _get_config_file(model_path, model_name):
        # Name of the file for storing hyperparameter details
        return os.path.join(model_path, model_name + ".config")

def _get_model_file(model_path, model_name):
```

(continues on next page)

(continued from previous page)

```

# Name of the file for storing network parameters
return os.path.join(model_path, model_name + ".tar")

def load_model(model_path, model_name, net=None):
    """
    Loads a saved model from disk.

    Inputs:
        model_path - Path of the checkpoint directory
        model_name - Name of the model (str)
        net - (Optional) If given, the state dict is loaded into this model. Otherwise,
    ↪ a new model is created.
    """

    config_file, model_file = _get_config_file(model_path, model_name), _get_model_
    ↪ file(model_path, model_name)
    assert os.path.isfile(config_file), f"Could not find the config file \\"{config_file}\\".
    ↪ . Are you sure this is the correct path and you have your model config stored here?"
    assert os.path.isfile(model_file), f"Could not find the model file \\"{model_file}\\".
    ↪ Are you sure this is the correct path and you have your model stored here?"
    with open(config_file, "r") as f:
        config_dict = json.load(f)
    if net is None:
        act_fn_name = config_dict["act_fn"].pop("name").lower()
        act_fn = act_fn_by_name[act_fn_name](**config_dict.pop("act_fn"))
        net = BaseNetwork(act_fn=act_fn, **config_dict)
        net.load_state_dict(torch.load(model_file, map_location=device))
    return net

def save_model(model, model_path, model_name):
    """
    Given a model, we save the state_dict and hyperparameters.

    Inputs:
        model - Network object to save parameters from
        model_path - Path of the checkpoint directory
        model_name - Name of the model (str)
    """

    config_dict = model.config
    os.makedirs(model_path, exist_ok=True)
    config_file, model_file = _get_config_file(model_path, model_name), _get_model_
    ↪ file(model_path, model_name)
    with open(config_file, "w") as f:
        json.dump(config_dict, f)
    torch.save(model.state_dict(), model_file)

```

We also set up the dataset we want to train it on, namely [FashionMNIST](#). FashionMNIST is a more complex version of MNIST and contains black-and-white images of clothes instead of digits. The 10 classes include trousers, coats, shoes, bags and more. To load this dataset, we will make use of yet another PyTorch package, namely [torchvision](#) ([documentation](#)). The [torchvision](#) package consists of popular datasets, model architectures, and common image transformations for computer vision. We will use the package for many of the notebooks in this course to simplify our dataset handling.

Let's load the dataset below, and visualize a few images to get an impression of the data.

```
[12]: import torchvision
from torchvision.datasets import FashionMNIST
from torchvision import transforms

# Transformations applied on each image => first make them a tensor, then normalize them
# in the range -1 to 1
transform = transforms.Compose([transforms.ToTensor(),
                               transforms.Normalize((0.5,), (0.5,))])

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = FashionMNIST(root=DATASET_PATH, train=True, transform=transform,
                             download=True)
train_set, val_set = torch.utils.data.random_split(train_dataset, [50000, 10000])

# Loading the test set
test_set = FashionMNIST(root=DATASET_PATH, train=False, transform=transform,
                         download=True)

# We define a set of data loaders that we can use for various purposes later.
# Note that for actually training a model, we will use different data loaders
# with a lower batch size.
train_loader = data.DataLoader(train_set, batch_size=1024, shuffle=True, drop_last=False)
val_loader = data.DataLoader(val_set, batch_size=1024, shuffle=False, drop_last=False)
test_loader = data.DataLoader(test_set, batch_size=1024, shuffle=False, drop_last=False)
```

```
[13]: exmp_imgs = [train_set[i][0] for i in range(16)]
# Organize the images into a grid for nicer visualization
img_grid = torchvision.utils.make_grid(torch.stack(exmp_imgs, dim=0), nrow=4,
                                       normalize=True, pad_value=0.5)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("FashionMNIST examples")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

FashionMNIST examples



Visualizing the gradient flow after initialization

As mentioned previously, one important aspect of activation functions is how they propagate gradients through the network. Imagine we have a very deep neural network with more than 50 layers. The gradients for the input layer, i.e. the very first layer, have passed >50 times the activation function, but we still want them to be of a reasonable size. If the gradient through the activation function is (in expectation) considerably smaller than 1, our gradients will vanish until they reach the input layer. If the gradient through the activation function is larger than 1, the gradients exponentially increase and might explode.

To get a feeling of how every activation function influences the gradients, we can look at a freshly initialized network and measure the gradients for each parameter for a batch of 256 images:

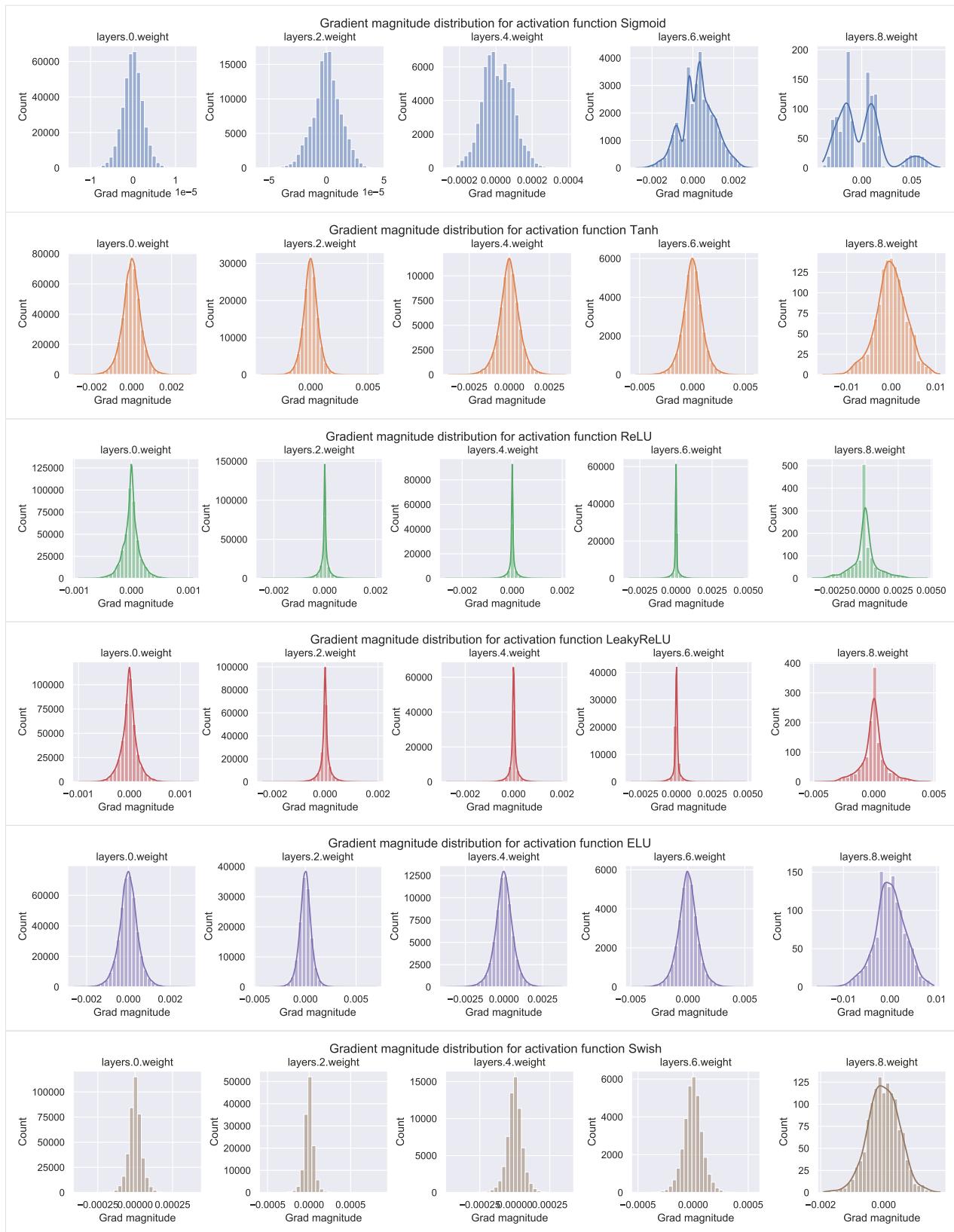
```
[14]: def visualize_gradients(net, color="C0"):
    """
    Inputs:
        net - Object of class BaseNetwork
        color - Color in which we want to visualize the histogram (for easier separation
    ↪of activation functions)
    """

    net.eval()
    small_loader = data.DataLoader(train_set, batch_size=256, shuffle=False)
    imgs, labels = next(iter(small_loader))
    imgs, labels = imgs.to(device), labels.to(device)

    # Pass one batch through the network, and calculate the gradients for the weights
    net.zero_grad()
    preds = net(imgs)
    loss = F.cross_entropy(preds, labels)
    loss.backward()
    # We limit our visualization to the weight parameters and exclude the bias to reduce
    ↪the number of plots
    grads = {name: params.grad.data.view(-1).cpu().clone().numpy() for name, params in
    ↪net.named_parameters() if "weight" in name}
    net.zero_grad()

    ## Plotting
    columns = len(grads)
    fig, ax = plt.subplots(1, columns, figsize=(columns*3.5, 2.5))
    fig_index = 0
    for key in grads:
        key_ax = ax[fig_index%columns]
        sns.histplot(data=grads[key], bins=30, ax=key_ax, color=color, kde=True)
        key_ax.set_title(str(key))
        key_ax.set_xlabel("Grad magnitude")
        fig_index += 1
    fig.suptitle(f"Gradient magnitude distribution for activation function {net.config[
    ↪'act_fn'][['name']]}", fontsize=14, y=1.05)
    fig.subplots_adjust(wspace=0.45)
    plt.show()
    plt.close()
```

```
[15]: # Seaborn prints warnings if histogram has small values. We can ignore them for now
import warnings
warnings.filterwarnings('ignore')
## Create a plot for every activation function
for i, act_fn_name in enumerate(act_fn_by_name):
    set_seed(42) # Setting the seed ensures that we have the same weight initialization
    ↪for each activation function
    act_fn = act_fn_by_name[act_fn_name]()
    net_actfn = BaseNetwork(act_fn=act_fn).to(device)
    visualize_gradients(net_actfn, color=f"C{i}")
```



The sigmoid activation function shows a clearly undesirable behavior. While the gradients for the output layer are very large with up to 0.1, the input layer has the lowest gradient norm across all activation functions with only 10^{-5} . This

is due to its small maximum gradient of 1/4, and finding a suitable learning rate across all layers is not possible in this setup. All the other activation functions show to have similar gradient norms across all layers. Interestingly, the ReLU activation has a spike around 0 which is caused by its zero-part on the left, and dead neurons (we will take a closer look at this later on).

Note that additionally to the activation, the initialization of the weight parameters can be crucial. By default, PyTorch uses the [Kaiming](#) initialization for linear layers optimized for ReLU activations. In Tutorial 4, we will take a closer look at initialization, but assume for now that the Kaiming initialization works for all activation functions reasonably well.

Training a model

Next, we want to train our model with different activation functions on FashionMNIST and compare the gained performance. All in all, our final goal is to achieve the best possible performance on a dataset of our choice. Therefore, we write a training loop in the next cell including a validation after every epoch and a final test on the best model:

```
[16]: def train_model(net, model_name, max_epochs=50, patience=7, batch_size=256,_
    ↪overwrite=False):
    """
    Train a model on the training set of FashionMNIST

    Inputs:
        net - Object of BaseNetwork
        model_name - (str) Name of the model, used for creating the checkpoint names
        max_epochs - Number of epochs we want to (maximally) train for
        patience - If the performance on the validation set has not improved for
        ↪#patience epochs, we stop training early
        batch_size - Size of batches used in training
        overwrite - Determines how to handle the case when there already exists a
        ↪checkpoint. If True, it will be overwritten. Otherwise, we skip training.
    """

    file_exists = os.path.isfile(_get_model_file(CHECKPOINT_PATH, model_name))
    if file_exists and not overwrite:
        print("Model file already exists. Skipping training...")
    else:
        if file_exists:
            print("Model file exists, but will be overwritten...")

        # Defining optimizer, loss and data loader
        optimizer = optim.SGD(net.parameters(), lr=1e-2, momentum=0.9) # Default
        ↪parameters, feel free to change
        loss_module = nn.CrossEntropyLoss()
        train_loader_local = data.DataLoader(train_set, batch_size=batch_size,_
        ↪shuffle=True, drop_last=True, pin_memory=True)

        val_scores = []
        best_val_epoch = -1
        for epoch in range(max_epochs):
            #####
            # Training #
            #####
            net.train()
            true_preds, count = 0., 0
```

(continues on next page)

(continued from previous page)

```

    for imgs, labels in tqdm(train_loader_local, desc=f"Epoch {epoch+1}", ↵
    ↵leave=False):
        imgs, labels = imgs.to(device), labels.to(device) # To GPU
        optimizer.zero_grad() # Zero-grad can be placed anywhere before "loss.
    ↵backward() "
        preds = net(imgs)
        loss = loss_module(preds, labels)
        loss.backward()
        optimizer.step()
        # Record statistics during training
        true_preds += (preds.argmax(dim=-1) == labels).sum()
        count += labels.shape[0]
        train_acc = true_preds / count

#####
# Validation #
#####
val_acc = test_model(net, val_loader)
val_scores.append(val_acc)
print(f"[Epoch {epoch+1:2d}] Training accuracy: {train_acc*100.0:05.2f}%, ↵
    ↵Validation accuracy: {val_acc*100.0:05.2f}%")

if len(val_scores) == 1 or val_acc > val_scores[best_val_epoch]:
    print("\t (New best performance, saving model...)")
    save_model(net, CHECKPOINT_PATH, model_name)
    best_val_epoch = epoch
elif best_val_epoch <= epoch - patience:
    print(f'Early stopping due to no improvement over the last {patience} ↵
    ↵epochs')
    break

# Plot a curve of the validation accuracy
plt.plot([i for i in range(1,len(val_scores)+1)], val_scores)
plt.xlabel("Epochs")
plt.ylabel("Validation accuracy")
plt.title(f"Validation performance of {model_name}")
plt.show()
plt.close()

load_model(CHECKPOINT_PATH, model_name, net=net)
test_acc = test_model(net, test_loader)
print((f" Test accuracy: {test_acc*100.0:4.2f}% ").center(50, "=")+"\n")
return test_acc

def test_model(net, data_loader):
    """
    Test a model on a specified dataset.

    Inputs:
        net - Trained model of type BaseNetwork
        data_loader - DataLoader object of the dataset to test on (validation or test)
    """

```

(continues on next page)

(continued from previous page)

```
.....
net.eval()
true_preds, count = 0., 0
for imgs, labels in data_loader:
    imgs, labels = imgs.to(device), labels.to(device)
    with torch.no_grad():
        preds = net(imgs).argmax(dim=-1)
        true_preds += (preds == labels).sum().item()
        count += labels.shape[0]
test_acc = true_preds / count
return test_acc
```

We train one model for each activation function. We recommend using the pretrained models to save time if you are running this notebook on CPU.

```
[17]: for act_fn_name in act_fn_by_name:
    print(f"Training BaseNetwork with {act_fn_name} activation...")
    set_seed(42)
    act_fn = act_fn_by_name[act_fn_name]()
    net_actfn = BaseNetwork(act_fn=act_fn).to(device)
    train_model(net_actfn, f"FashionMNIST_{act_fn_name}", overwrite=False)

Training BaseNetwork with sigmoid activation...
Model file already exists. Skipping training...
===== Test accuracy: 10.00% =====

Training BaseNetwork with tanh activation...
Model file already exists. Skipping training...
===== Test accuracy: 87.59% =====

Training BaseNetwork with relu activation...
Model file already exists. Skipping training...
===== Test accuracy: 88.62% =====

Training BaseNetwork with leakyrelu activation...
Model file already exists. Skipping training...
===== Test accuracy: 88.92% =====

Training BaseNetwork with elu activation...
Model file already exists. Skipping training...
===== Test accuracy: 87.27% =====

Training BaseNetwork with swish activation...
Model file already exists. Skipping training...
===== Test accuracy: 88.73% =====
```

Not surprisingly, the model using the sigmoid activation function shows to fail and does not improve upon random performance (10 classes => 1/10 for random chance).

All the other activation functions gain similar performance. To have a more accurate conclusion, we would have to train the models for multiple seeds and look at the averages. However, the “optimal” activation function also depends on many other factors (hidden sizes, number of layers, type of layers, task, dataset, optimizer, learning rate, etc.) so that a thorough grid search would not be useful in our case. In the literature, activation functions that have shown to

work well with deep networks are all types of ReLU functions we experiment with here, with small gains for specific activation functions in specific networks.

Visualizing the activation distribution

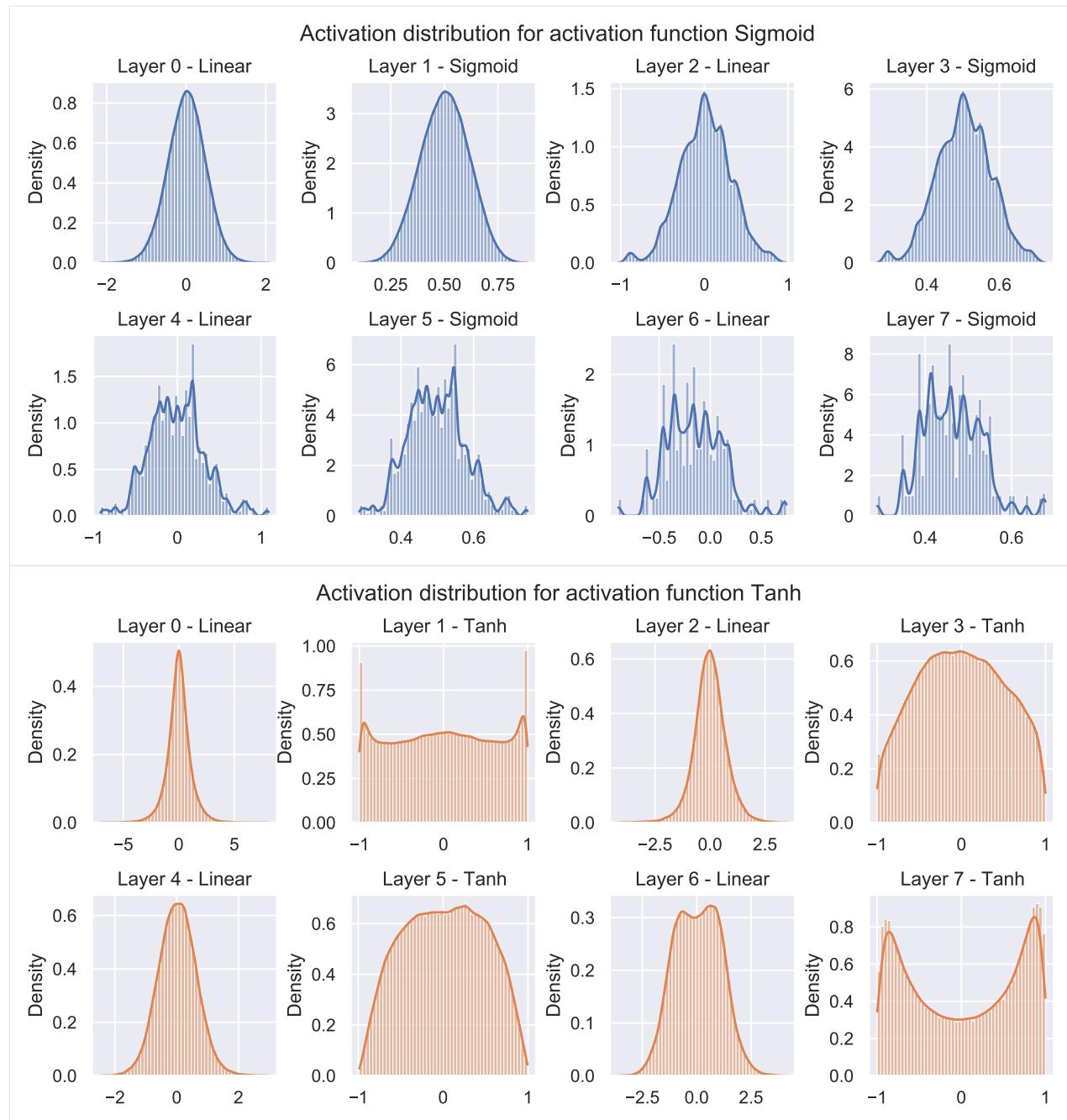
After we have trained the models, we can look at the actual activation values that find inside the model. For instance, how many neurons are set to zero in ReLU? Where do we find most values in Tanh? To answer these questions, we can write a simple function which takes a trained model, applies it to a batch of images, and plots the histogram of the activations inside the network:

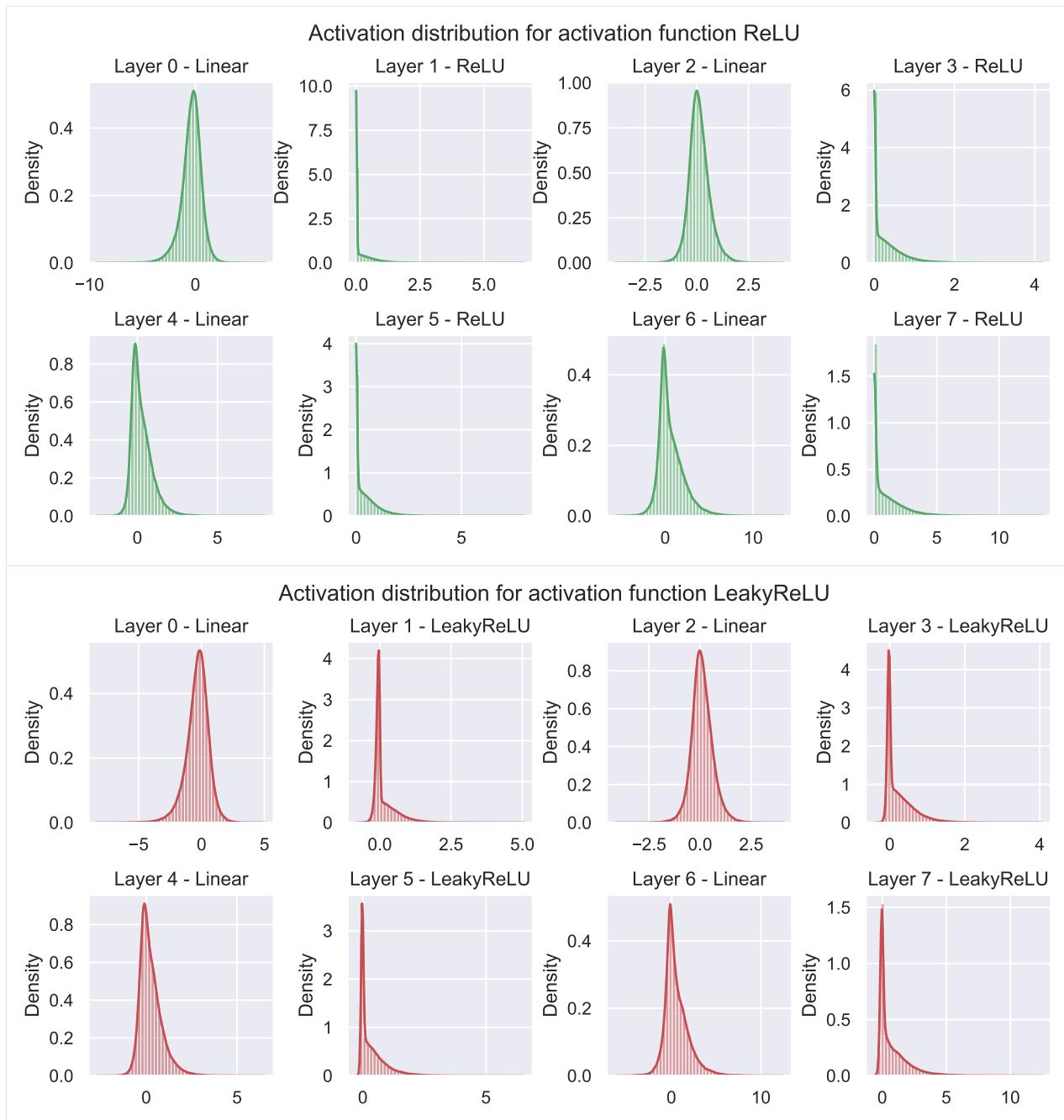
```
[18]: def visualize_activations(net, color="C0"):
    activations = {}

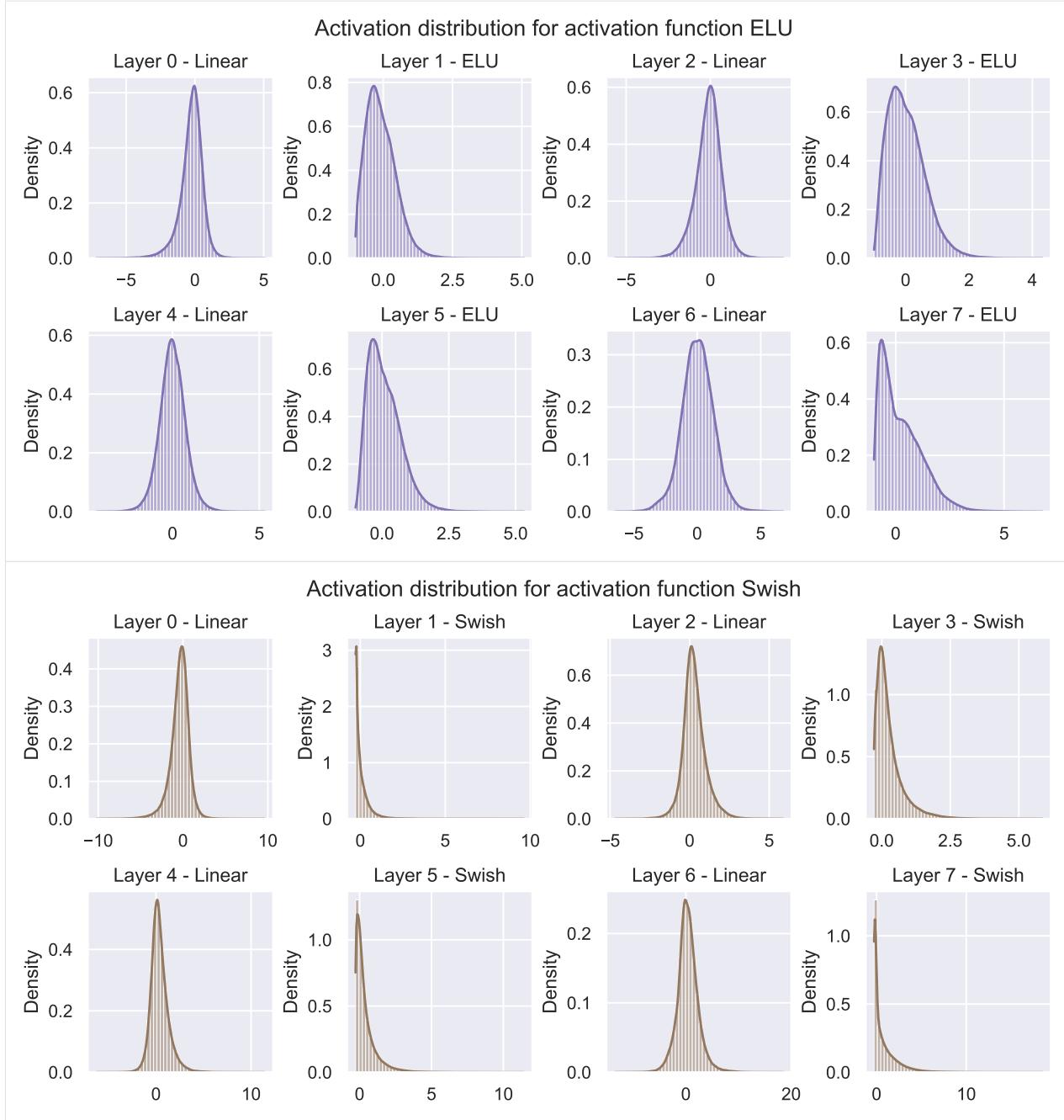
    net.eval()
    small_loader = data.DataLoader(train_set, batch_size=1024)
    imgs, labels = next(iter(small_loader))
    with torch.no_grad():
        layer_index = 0
        imgs = imgs.to(device)
        imgs = imgs.view(imgs.size(0), -1)
        # We need to manually loop through the layers to save all activations
        for layer_index, layer in enumerate(net.layers[:-1]):
            imgs = layer(imgs)
            activations[layer_index] = imgs.view(-1).cpu().numpy()

    ## Plotting
    columns = 4
    rows = math.ceil(len(activations)/columns)
    fig, ax = plt.subplots(rows, columns, figsize=(columns*2.7, rows*2.5))
    fig_index = 0
    for key in activations:
        key_ax = ax[fig_index//columns][fig_index%columns]
        sns.histplot(data=activations[key], bins=50, ax=key_ax, color=color, kde=True, stat="density")
        key_ax.set_title(f"Layer {key} - {net.layers[key].__class__.__name__}")
        fig_index += 1
    fig.suptitle("Activation distribution for activation function " + net.config['act_fn'][name], fontsize=14)
    fig.subplots_adjust(hspace=0.4, wspace=0.4)
    plt.show()
    plt.close()
```

```
[19]: for i, act_fn_name in enumerate(act_fn_by_name):
    net_actfn = load_model(model_path=CHECKPOINT_PATH, model_name=f"FashionMNIST_{act_fn_name}).to(device)
    visualize_activations(net_actfn, color=f"C{i}")
```







As the model with sigmoid activation was not able to train properly, the activations are also less informative and all gathered around 0.5 (the activation at input 0).

The tanh shows a more diverse behavior. While for the input layer we experience a larger amount of neurons to be close to -1 and 1, where the gradients are close to zero, the activations in the two consecutive layers are closer to zero. This is probably because the input layers look for specific features in the input image, and the consecutive layers combine those together. The activations for the last layer are again more biased to the extreme points because the classification layer can be seen as a weighted average of those values (the gradients push the activations to those extremes).

The ReLU has a strong peak at 0, as we initially expected. The effect of having no gradients for negative values is that the network does not have a Gaussian-like distribution after the linear layers, but a longer tail towards the positive values. The LeakyReLU shows a very similar behavior while ELU follows again a more Gaussian-like distribution.

The Swish activation seems to lie in between, although it is worth noting that Swish uses significantly higher values than other activation functions (up to 20).

As all activation functions show slightly different behavior although obtaining similar performance for our simple network, it becomes apparent that the selection of the “optimal” activation function really depends on many factors, and is not the same for all possible networks.

Finding dead neurons in ReLU networks

One known drawback of the ReLU activation is the occurrence of “dead neurons”, i.e. neurons with no gradient for any training input. The issue of dead neurons is that as no gradient is provided for the layer, we cannot train the parameters of this neuron in the previous layer to obtain output values besides zero. For dead neurons to happen, the output value of a specific neuron of the linear layer before the ReLU has to be negative for all input images. Considering the large number of neurons we have in a neural network, it is not unlikely for this to happen.

To get a better understanding of how much of a problem this is, and when we need to be careful, we will measure how many dead neurons different networks have. For this, we implement a function which runs the network on the whole training set and records whether a neuron is exactly 0 for all data points or not:

```
[20]: def measure_number_dead_neurons(net):

    # For each neuron, we create a boolean variable initially set to 1. If it has an
    # activation unequal 0 at any time,
    # we set this variable to 0. After running through the whole training set, only dead
    # neurons will have a 1.
    neurons_dead = [
        torch.ones(layer.weight.shape[0], device=device, dtype=torch.bool) for layer in
        net.layers[:-1] if isinstance(layer, nn.Linear)
    ] # Same shapes as hidden size in BaseNetwork

    net.eval()
    with torch.no_grad():
        for imgs, labels in tqdm(train_loader, leave=False): # Run through whole
            #training set
                layer_index = 0
                imgs = imgs.to(device)
                imgs = imgs.view(imgs.size(0), -1)
                for layer in net.layers[:-1]:
                    imgs = layer(imgs)
                    if isinstance(layer, ActivationFunction):
                        # Are all activations == 0 in the batch, and we did not record the
                        # opposite in the last batches?
                        neurons_dead[layer_index] = torch.logical_and(neurons_dead[layer_
                        index], (imgs == 0).all(dim=0))
                    layer_index += 1
                number_neurons_dead = [t.sum().item() for t in neurons_dead]
                print("Number of dead neurons:", number_neurons_dead)
                print("In percentage:", ", ".join([f"{100.0 * num_dead / tens.shape[0]):4.2f}%" for_
                tens, num_dead in zip(neurons_dead, number_neurons_dead)]))
```

First, we can measure the number of dead neurons for an untrained network:

```
[21]: set_seed(42)
net_relu = BaseNetwork(act_fn=ReLU()).to(device)
```

(continues on next page)

(continued from previous page)

```
measure_number_dead_neurons(net_relu)
HBox(children=(FloatProgress(value=0.0, max=49.0), HTML(value='')))

Number of dead neurons: [0, 0, 3, 10]
In percentage: 0.00%, 0.00%, 1.17%, 7.81%
```

We see that only a minor amount of neurons are dead, but that they increase with the depth of the layer. However, this is not a problem for the small number of dead neurons we have as the input to later layers is changed due to updates to the weights of previous layers. Therefore, dead neurons in later layers can potentially become “alive”/active again.

How does this look like for a trained network (with the same initialization)?

```
[22]: net_relu = load_model(model_path=CHECKPOINT_PATH, model_name="FashionMNIST_relu").
        to(device)
measure_number_dead_neurons(net_relu)
HBox(children=(FloatProgress(value=0.0, max=49.0), HTML(value='')))

Number of dead neurons: [0, 0, 0, 3]
In percentage: 0.00%, 0.00%, 0.00%, 2.34%
```

The number of dead neurons indeed decreased in the later layers. However, it should be noted that dead neurons are especially problematic in the input layer. As the input does not change over epochs (the training set is kept as it is), training the network cannot turn those neurons back active. Still, the input data has usually a sufficiently high standard deviation to reduce the risk of dead neurons.

Finally, we check how the number of dead neurons behaves with increasing layer depth. For instance, let's take the following 10-layer neural network:

```
[23]: set_seed(42)
net_relu = BaseNetwork(act_fn=ReLU(), hidden_sizes=[256, 256, 256, 256, 256, 256, 128, 128,
        128, 128, 128]).to(device)
measure_number_dead_neurons(net_relu)
HBox(children=(FloatProgress(value=0.0, max=49.0), HTML(value='')))

Number of dead neurons: [0, 0, 7, 27, 89, 60, 58, 61, 72, 56]
In percentage: 0.00%, 0.00%, 2.73%, 10.55%, 34.77%, 46.88%, 45.31%, 47.66%, 56.25%, 43.75
        %
```

The number of dead neurons is significantly higher than before which harms the gradient flow especially in the first iterations. For instance, more than 56% of the neurons in the pre-last layer are dead which creates a considerable bottleneck. Hence, it is advisable to use other nonlinearities like Swish for very deep networks.

4.6.3 Conclusion

In this notebook, we have reviewed a set of six activation functions (sigmoid, tanh, ReLU, LeakyReLU, ELU, and Swish) in neural networks, and discussed how they influence the gradient distribution across layers. Sigmoid tends to fail deep neural networks as the highest gradient it provides is 0.25 leading to vanishing gradients in early layers. All ReLU-based activation functions have shown to perform well, and besides the original ReLU, do not have the issue of dead neurons. When implementing your own neural network, it is recommended to start with a ReLU-based network and select the specific activation function based on the properties of the network.

4.6.4 References

[1] Ramachandran, Prajit, Barret Zoph, and Quoc V. Le. “Searching for activation functions.” arXiv preprint arXiv:1710.05941 (2017). [Paper link](#)

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.7 Tutorial 4: Optimization and Initialization

Filled notebook:

Pre-trained models:

Recordings:

JAX+Flax version:

Author: Phillip Lippe

Note: Interested in JAX? Check out our [JAX+Flax version](#) of this tutorial!

In this tutorial, we will review techniques for optimization and initialization of neural networks. When increasing the depth of neural networks, there are various challenges we face. Most importantly, we need to have a stable gradient flow through the network, as otherwise, we might encounter vanishing or exploding gradients. This is why we will take a closer look at the following concepts: initialization and optimization.

In the first half of the notebook, we will review different initialization techniques, and go step by step from the simplest initialization to methods that are nowadays used in very deep networks. In the second half, we focus on optimization comparing the optimizers SGD, SGD with Momentum, and Adam.

Let’s start with importing our standard libraries:

```
[1]: ## Standard libraries
import os
import json
import math
import numpy as np
import copy

## Imports for plotting
import matplotlib.pyplot as plt
from matplotlib import cm
%matplotlib inline
from IPython.display import set_matplotlib_formats
```

(continues on next page)

(continued from previous page)

```
set_matplotlib_formats('svg', 'pdf') # For export
import seaborn as sns
sns.set()

## Progress bar
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
```

We will use the same `set_seed` function as in Tutorial 3, as well as the path variables `DATASET_PATH` and `CHECKPOINT_PATH`. Adjust the paths if necessary.

```
[2]: # Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial4"

# Function for setting the seed
def set_seed(seed):
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
set_seed(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Fetching the device that will be used throughout this notebook
device = torch.device("cpu") if not torch.cuda.is_available() else torch.device("cuda:0")
print("Using device", device)
```

Using device cuda:0

In the last part of the notebook, we will train models using three different optimizers. The pretrained models for those are downloaded below.

```
[3]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial4/"
# Files to download
pretrained_files = ["FashionMNIST_SGD.config",      "FashionMNIST_SGD_results.json",
                    "FashionMNIST_SGD.tar",
                    "FashionMNIST_SGDMom.config", "FashionMNIST_SGDMom_results.json",
                    "FashionMNIST_SGDMom.tar", ]
```

(continues on next page)

(continued from previous page)

```

    "FashionMNIST_Adam.config",    "FashionMNIST_Adam_results.json",
→ "FashionMNIST_Adam.tar"    ]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive_"
→ folder, or contact the author with the full output including the following error:\n",_
→ e)

```

4.7.1 Preparation

Throughout this notebook, we will use a deep fully connected network, similar to our previous tutorial. We will also again apply the network to FashionMNIST, so you can relate to the results of Tutorial 3. We start by loading the FashionMNIST dataset:

```
[4]: from torchvision.datasets import FashionMNIST
from torchvision import transforms

# Transformations applied on each image => first make them a tensor, then normalize them_
→ with mean 0 and std 1
transform = transforms.Compose([transforms.ToTensor(),
                               transforms.Normalize((0.2861,), (0.3530,))])
                           ])

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = FashionMNIST(root=DATASET_PATH, train=True, transform=transform,_
→ download=True)
train_set, val_set = torch.utils.data.random_split(train_dataset, [50000, 10000])

# Loading the test set
test_set = FashionMNIST(root=DATASET_PATH, train=False, transform=transform,_
→ download=True)

# We define a set of data loaders that we can use for various purposes later.
# Note that for actually training a model, we will use different data loaders
# with a lower batch size.
train_loader = data.DataLoader(train_set, batch_size=1024, shuffle=True, drop_last=False)
val_loader = data.DataLoader(val_set, batch_size=1024, shuffle=False, drop_last=False)
test_loader = data.DataLoader(test_set, batch_size=1024, shuffle=False, drop_last=False)
```

In comparison to the previous tutorial, we have changed the parameters of the normalization transformation `transforms.Normalize`. The normalization is now designed to give us an expected mean of 0 and a standard deviation of 1 across pixels. This will be particularly relevant for the discussion about initialization we will look at below,

and hence we change it here. It should be noted that in most classification tasks, both normalization techniques (between -1 and 1 or mean 0 and stddev 1) have shown to work well. We can calculate the normalization parameters by determining the mean and standard deviation on the original images:

```
[5]: print("Mean", (train_dataset.data.float() / 255.0).mean().item())
print("Std", (train_dataset.data.float() / 255.0).std().item())

Mean 0.2860923707485199
Std 0.3530242443084717
```

We can verify the transformation by looking at the statistics of a single batch:

```
[6]: imgs, _ = next(iter(train_loader))
print(f"Mean: {imgs.mean().item():.3f}")
print(f"Standard deviation: {imgs.std().item():.3f}")
print(f"Maximum: {imgs.max().item():.3f}")
print(f"Minimum: {imgs.min().item():.3f}")

Mean: 0.002
Standard deviation: 1.001
Maximum: 2.022
Minimum: -0.810
```

Note that the maximum and minimum are not 1 and -1 anymore, but shifted towards the positive values. This is because FashionMNIST contains a lot of black pixels, similar to MNIST.

Next, we create a linear neural network. We use the same setup as in the previous tutorial.

```
[7]: class BaseNetwork(nn.Module):

    def __init__(self, act_fn, input_size=784, num_classes=10, hidden_sizes=[512, 256, 256, 128]):
        """
        Inputs:
            act_fn - Object of the activation function that should be used as non-linearity in the network.
            input_size - Size of the input images in pixels
            num_classes - Number of classes we want to predict
            hidden_sizes - A list of integers specifying the hidden layer sizes in the NN
        """
        super().__init__()

        # Create the network based on the specified hidden sizes
        layers = []
        layer_sizes = [input_size] + hidden_sizes
        for layer_index in range(1, len(layer_sizes)):
            layers += [nn.Linear(layer_sizes[layer_index-1], layer_sizes[layer_index]), act_fn]
        layers += [nn.Linear(layer_sizes[-1], num_classes)]
        self.layers = nn.ModuleList(layers) # A module list registers a list of modules as submodules (e.g. for parameters)

        self.config = {"act_fn": act_fn.__class__.__name__, "input_size": input_size,
                      "num_classes": num_classes, "hidden_sizes": hidden_sizes}
```

(continues on next page)

(continued from previous page)

```
def forward(self, x):
    x = x.view(x.size(0), -1)
    for l in self.layers:
        x = l(x)
    return x
```

For the activation functions, we make use of PyTorch's `torch.nn` library instead of implementing ourselves. However, we also define an `Identity` activation function. Although this activation function would significantly limit the network's modeling capabilities, we will use it in the first steps of our discussion about initialization (for simplicity).

```
[8]: class Identity(nn.Module):
    def forward(self, x):
        return x

act_fn_by_name = {
    "tanh": nn.Tanh,
    "relu": nn.ReLU,
    "identity": Identity
}
```

Finally, we define a few plotting functions that we will use for our discussions. These functions help us to (1) visualize the weight/parameter distribution inside a network, (2) visualize the gradients that the parameters at different layers receive, and (3) the activations, i.e. the output of the linear layers. The detailed code is not important, but feel free to take a closer look if interested.

```
[9]: #####
def plot_dists(val_dict, color="C0", xlabel=None, stat="count", use_kde=True):
    columns = len(val_dict)
    fig, ax = plt.subplots(1, columns, figsize=(columns*3, 2.5))
    fig_index = 0
    for key in sorted(val_dict.keys()):
        key_ax = ax[fig_index%columns]
        sns.histplot(val_dict[key], ax=key_ax, color=color, bins=50, stat=stat,
                     kde=use_kde and ((val_dict[key].max()-val_dict[key].min())>1e-8)) #
        ←Only plot kde if there is variance
        key_ax.set_title(f'{key} ' + (r'(%i $\to$ %i)' % (val_dict[key].shape[1], val_
        ←dict[key].shape[0])) if len(val_dict[key].shape)>1 else ""))
        if xlabel is not None:
            key_ax.set_xlabel(xlabel)
        fig_index += 1
    fig.subplots_adjust(wspace=0.4)
    return fig

#####
def visualize_weight_distribution(model, color="C0"):
    weights = []
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            continue
        key_name = f"Layer {name.split('.')[1]}"
```

(continues on next page)

(continued from previous page)

```

weights[key_name] = param.detach().view(-1).cpu().numpy()

## Plotting
fig = plot_dists(weights, color=color, xlabel="Weight vals")
fig.suptitle("Weight distribution", fontsize=14, y=1.05)
plt.show()
plt.close()

#####
#def visualize_gradients(model, color="C0", print_variance=False):
#    """
#        Inputs:
#            net - Object of class BaseNetwork
#            color - Color in which we want to visualize the histogram (for easier separation
#                    of activation functions)
#    """
#    model.eval()
#    small_loader = data.DataLoader(train_set, batch_size=1024, shuffle=False)
#    imgs, labels = next(iter(small_loader))
#    imgs, labels = imgs.to(device), labels.to(device)

#    # Pass one batch through the network, and calculate the gradients for the weights
#    model.zero_grad()
#    preds = model(imgs)
#    loss = F.cross_entropy(preds, labels) # Same as nn.CrossEntropyLoss, but as a
#    #function instead of module
#    loss.backward()
#    # We limit our visualization to the weight parameters and exclude the bias to reduce
#    #the number of plots
#    grads = {name: params.grad.view(-1).cpu().clone().numpy() for name, params in model.
#    #named_parameters() if "weight" in name}
#    model.zero_grad()

#    ## Plotting
#    fig = plot_dists(grads, color=color, xlabel="Grad magnitude")
#    fig.suptitle("Gradient distribution", fontsize=14, y=1.05)
#    plt.show()
#    plt.close()

#    if print_variance:
#        for key in sorted(grads.keys()):
#            print(f"{key} - Variance: {np.var(grads[key])}")

#####
#def visualize_activations(model, color="C0", print_variance=False):
#    model.eval()
#    small_loader = data.DataLoader(train_set, batch_size=1024, shuffle=False)
#    imgs, labels = next(iter(small_loader))
#    imgs, labels = imgs.to(device), labels.to(device)

```

(continues on next page)

(continued from previous page)

```

# Pass one batch through the network, and calculate the gradients for the weights
feats = imgs.view(imgs.shape[0], -1)
activations = {}
with torch.no_grad():
    for layer_index, layer in enumerate(model.layers):
        feats = layer(feats)
        if isinstance(layer, nn.Linear):
            activations[f"Layer {layer_index}"] = feats.view(-1).detach().cpu().
numpy()

## Plotting
fig = plot_dists(activations, color=color, stat="density", xlabel="Activation vals")
fig.suptitle("Activation distribution", fontsize=14, y=1.05)
plt.show()
plt.close()

if print_variance:
    for key in sorted(activations.keys()):
        print(f"{key} - Variance: {np.var(activations[key])}")

#####

```

4.7.2 Initialization

Before starting our discussion about initialization, it should be noted that there exist many very good blog posts about the topic of neural network initialization (for example [deeplearning.ai](#), or a more [math-focused blog post](#)). In case something remains unclear after this tutorial, we recommend skimming through these blog posts as well.

When initializing a neural network, there are a few properties we would like to have. First, the variance of the input should be propagated through the model to the last layer, so that we have a similar standard deviation for the output neurons. If the variance would vanish the deeper we go in our model, it becomes much harder to optimize the model as the input to the next layer is basically a single constant value. Similarly, if the variance increases, it is likely to explode (i.e. head to infinity) the deeper we design our model. The second property we look out for in initialization techniques is a gradient distribution with equal variance across layers. If the first layer receives much smaller gradients than the last layer, we will have difficulties in choosing an appropriate learning rate.

As a starting point for finding a good method, we will analyze different initialization based on our linear neural network with no activation function (i.e. an identity). We do this because initializations depend on the specific activation function used in the network, and we can adjust the initialization schemes later on for our specific choice.

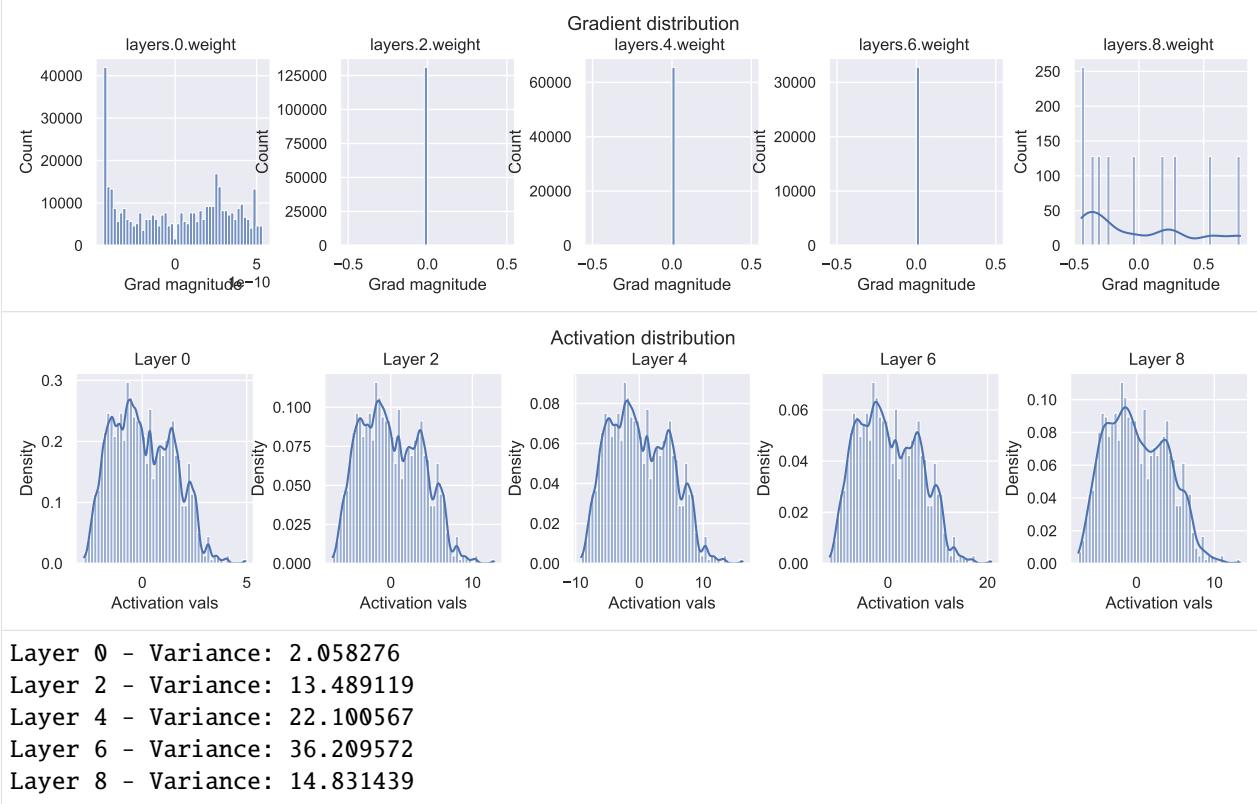
```
[10]: model = BaseNetwork(act_fn=Identity()).to(device)
```

Constant initialization

The first initialization we can consider is to initialize all weights with the same constant value. Intuitively, setting all weights to zero is not a good idea as the propagated gradient will be zero. However, what happens if we set all weights to a value slightly larger or smaller than 0? To find out, we can implement a function for setting all parameters below and visualize the gradients.

```
[11]: def const_init(model, c=0.0):
    for name, param in model.named_parameters():
        param.data.fill_(c)

const_init(model, c=0.005)
visualize_gradients(model)
visualize_activations(model, print_variance=True)
```



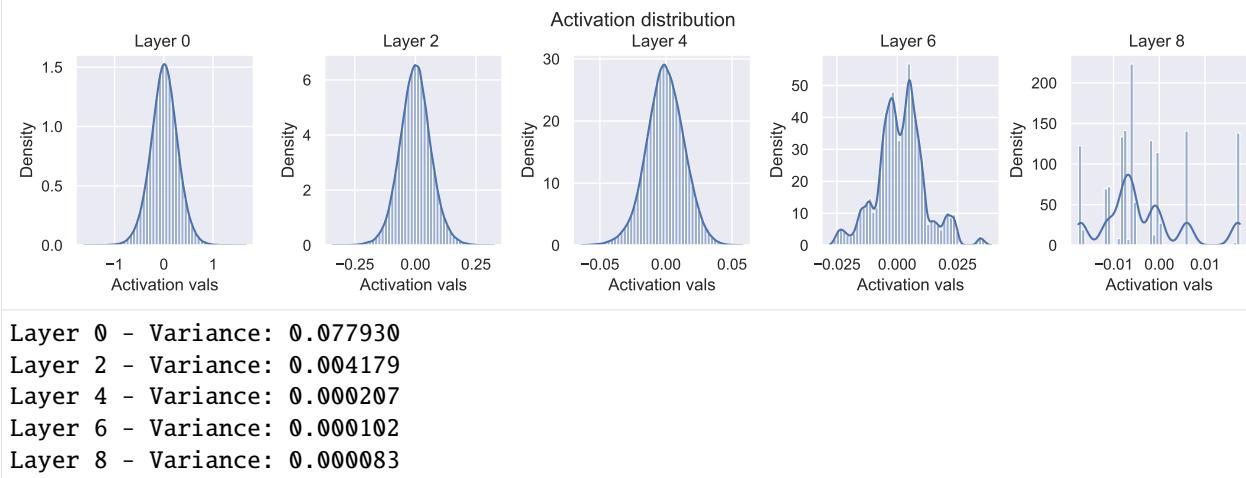
As we can see, only the first and the last layer have diverse gradient distributions while the other three layers have the same gradient for all weights (note that this value is unequal 0, but often very close to it). Having the same gradient for parameters that have been initialized with the same values means that we will always have the same value for those parameters. This would make our layer useless and reduce our effective number of parameters to 1. Thus, we cannot use a constant initialization to train our networks.

Constant variance

From the experiment above, we have seen that a constant value is not working. So instead, how about we initialize the parameters by randomly sampling from a distribution like a Gaussian? The most intuitive way would be to choose one variance that is used for all layers in the network. Let's implement it below, and visualize the activation distribution across layers.

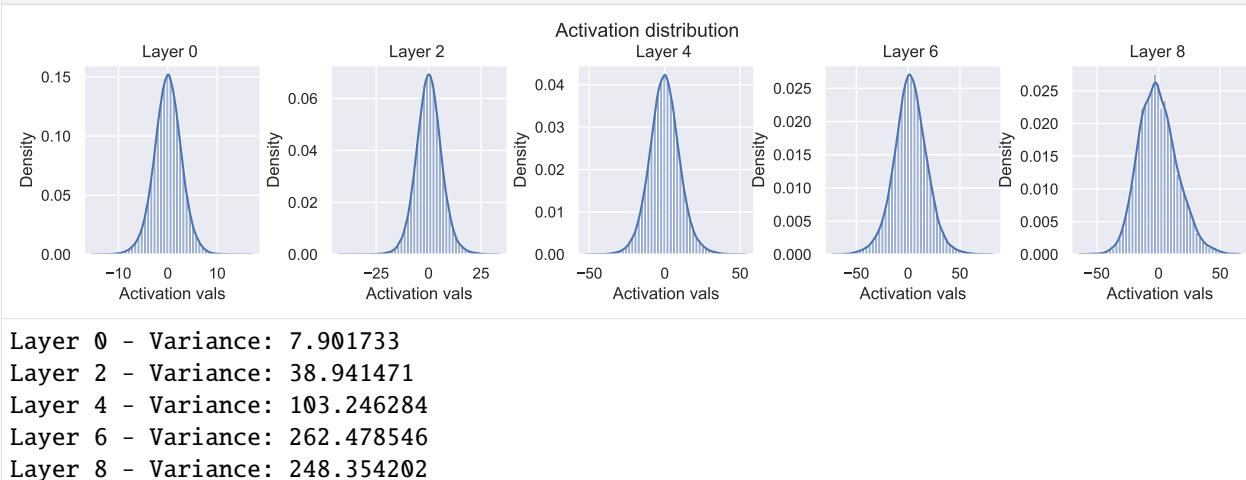
```
[12]: def var_init(model, std=0.01):
    for name, param in model.named_parameters():
        param.data.normal_(std=std)

var_init(model, std=0.01)
visualize_activations(model, print_variance=True)
```



The variance of the activation becomes smaller and smaller across layers, and almost vanishes in the last layer. Alternatively, we could use a higher standard deviation:

```
[13]: var_init(model, std=0.1)
visualize_activations(model, print_variance=True)
```



With a higher standard deviation, the activations are likely to explode. You can play around with the specific standard deviation values, but it will be hard to find one that gives us a good activation distribution across layers and is very specific to our model. If we would change the hidden sizes or number of layers, you would have to search all over again, which is neither efficient nor recommended.

How to find appropriate initialization values

From our experiments above, we have seen that we need to sample the weights from a distribution, but are not sure which one exactly. As a next step, we will try to find the optimal initialization from the perspective of the activation distribution. For this, we state two requirements:

1. The mean of the activations should be zero
2. The variance of the activations should stay the same across every layer

Suppose we want to design an initialization for the following layer: $y = Wx + b$ with $y \in \mathbb{R}^{d_y}$, $x \in \mathbb{R}^{d_x}$. Our goal is that the variance of each element of y is the same as the input, i.e. $\text{Var}(y_i) = \text{Var}(x_i) = \sigma_x^2$, and that the mean is zero. We assume x to also have a mean of zero, because, in deep neural networks, y would be the input of another layer. This requires the bias and weight to have an expectation of 0. Actually, as b is a single element per output neuron and is constant across different inputs, we set it to 0 overall.

Next, we need to calculate the variance with which we need to initialize the weight parameters. Along the calculation, we will need the following variance rule: given two independent variables, the variance of their product is $\text{Var}(X \cdot Y) = \mathbb{E}(Y)^2\text{Var}(X) + \mathbb{E}(X)^2\text{Var}(Y) + \text{Var}(X)\text{Var}(Y) = \mathbb{E}(Y^2)\mathbb{E}(X^2) - \mathbb{E}(Y)^2\mathbb{E}(X)^2$ (X and Y are not referring to x and y , but any random variable).

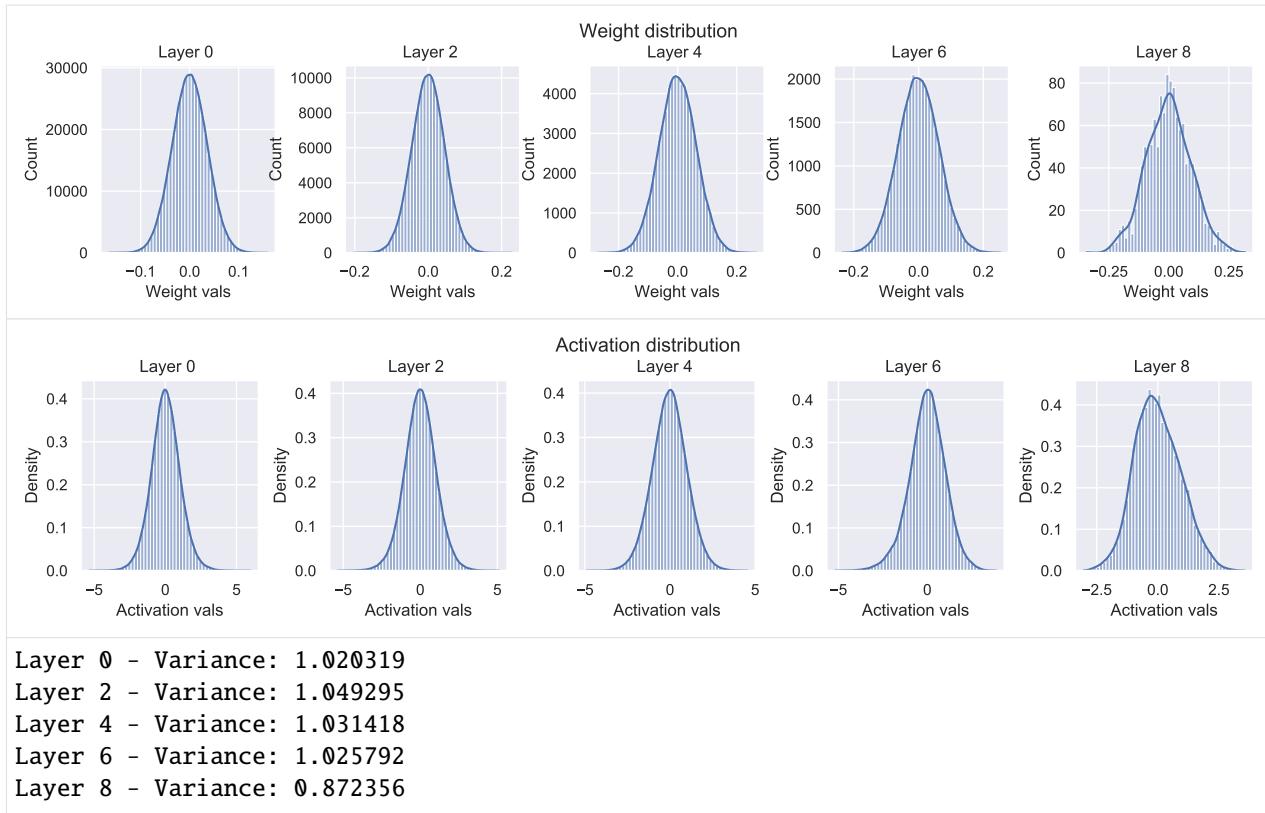
The needed variance of the weights, $\text{Var}(w_{ij})$, is calculated as follows:

$$\begin{aligned}
 y_i &= \sum_j w_{ij}x_j && \text{Calculation of a single output neuron without bias} \\
 \text{Var}(y_i) &= \sigma_x^2 = \text{Var}\left(\sum_j w_{ij}x_j\right) \\
 &= \sum_j \text{Var}(w_{ij}x_j) && \text{Inputs and weights are independent of each other} \\
 &= \sum_j \text{Var}(w_{ij}) \cdot \text{Var}(x_j) && \text{Variance rule (see above) with expectations being zero} \\
 &= d_x \cdot \text{Var}(w_{ij}) \cdot \text{Var}(x_j) && \text{Variance equal for all } d_x \text{ elements} \\
 &= \sigma_x^2 \cdot d_x \cdot \text{Var}(w_{ij}) \\
 \Rightarrow \text{Var}(w_{ij}) &= \sigma_W^2 = \frac{1}{d_x}
 \end{aligned}$$

Thus, we should initialize the weight distribution with a variance of the inverse of the input dimension d_x . Let's implement it below and check whether this holds:

```
[14]: def equal_var_init(model):
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            param.data.fill_(0)
        else:
            param.data.normal_(std=1.0/math.sqrt(param.shape[1]))

equal_var_init(model)
visualize_weight_distribution(model)
visualize_activations(model, print_variance=True)
```



As we expected, the variance stays indeed constant across layers. Note that our initialization does not restrict us to a normal distribution, but allows any other distribution with a mean of 0 and variance of $1/d_x$. You often see that a uniform distribution is used for initialization. A small benefit of using a uniform instead of a normal distribution is that we can exclude the chance of initializing very large or small weights.

Besides the variance of the activations, another variance we would like to stabilize is the one of the gradients. This ensures a stable optimization for deep networks. It turns out that we can do the same calculation as above starting from $\Delta x = W \Delta y$, and come to the conclusion that we should initialize our layers with $1/d_y$ where d_y is the number of output neurons. You can do the calculation as a practice, or check a thorough explanation in [this blog post](#). As a compromise between both constraints, [Glorot and Bengio \(2010\)](#) proposed to use the harmonic mean of both values. This leads us to the well-known Xavier initialization:

$$W \sim \mathcal{N} \left(0, \frac{2}{d_x + d_y} \right)$$

If we use a uniform distribution, we would initialize the weights with:

$$W \sim U \left[-\frac{\sqrt{6}}{\sqrt{d_x + d_y}}, \frac{\sqrt{6}}{\sqrt{d_x + d_y}} \right]$$

Let's shortly implement it and validate its effectiveness:

```
[15]: def xavier_init(model):
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            param.data.fill_(0)
        else:
            bound = math.sqrt(6)/math.sqrt(param.shape[0]+param.shape[1])
```

(continues on next page)

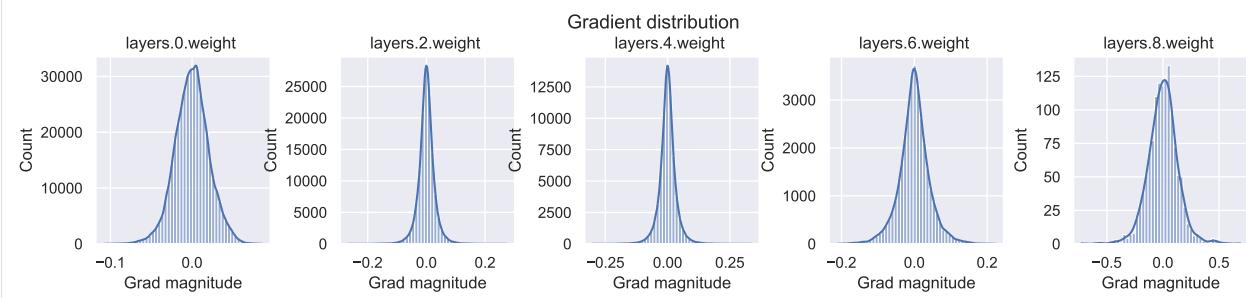
(continued from previous page)

```

param.data.uniform_(-bound, bound)

xavier_init(model)
visualize_gradients(model, print_variance=True)
visualize_activations(model, print_variance=True)

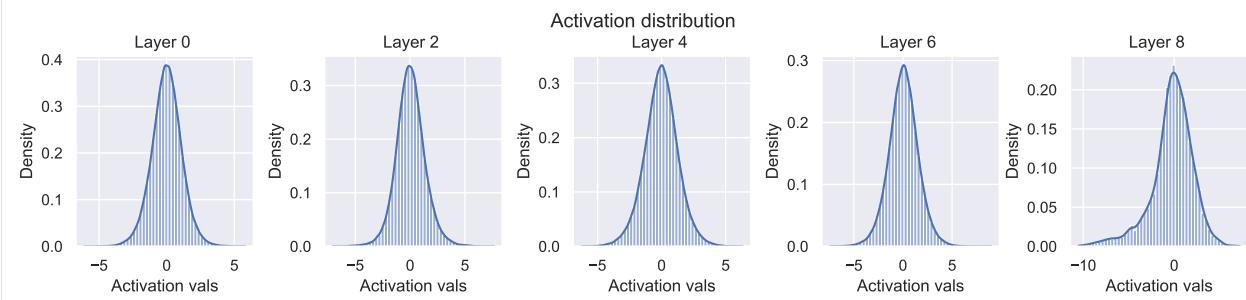
```



```

layer.0.weight - Variance: 0.000436
layer.2.weight - Variance: 0.000747
layer.4.weight - Variance: 0.001149
layer.6.weight - Variance: 0.001744
layer.8.weight - Variance: 0.017655

```



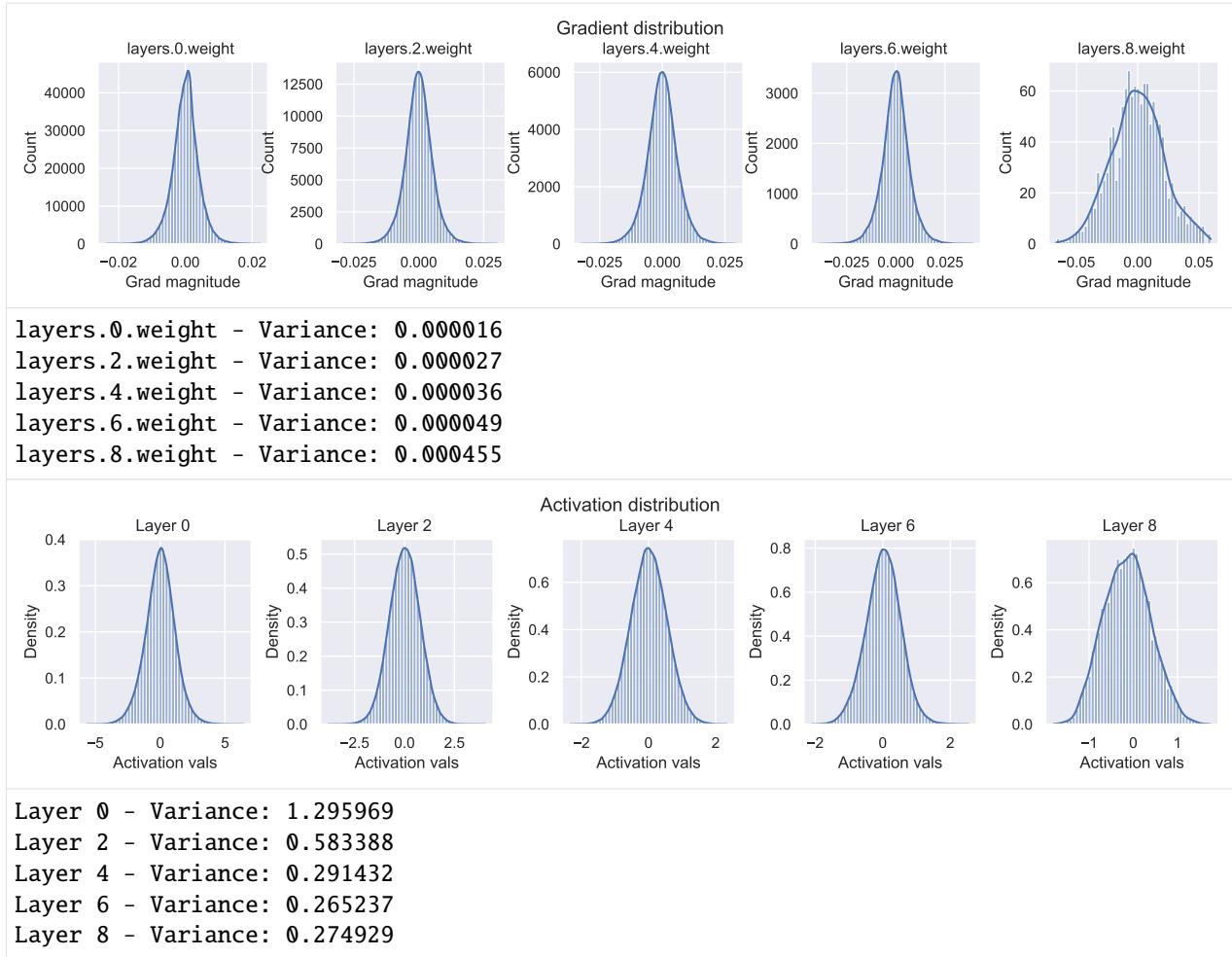
```

Layer 0 - Variance: 1.216592
Layer 2 - Variance: 1.719161
Layer 4 - Variance: 1.714506
Layer 6 - Variance: 2.224779
Layer 8 - Variance: 5.297660

```

We see that the Xavier initialization balances the variance of gradients and activations. Note that the significantly higher variance for the output layer is due to the large difference of input and output dimension (128 vs 10). However, we currently assumed the activation function to be linear. So what happens if we add a non-linearity? In a tanh-based network, a common assumption is that for small values during the initial steps in training, the tanh works as a linear function such that we don't have to adjust our calculation. We can check if that is the case for us as well:

```
[16]: model = BaseNetwork(act_fn=nn.Tanh()).to(device)
xavier_init(model)
visualize_gradients(model, print_variance=True)
visualize_activations(model, print_variance=True)
```



Although the variance decreases over depth, it is apparent that the activation distribution becomes more focused on the low values. Therefore, our variance will stabilize around 0.25 if we would go even deeper. Hence, we can conclude that the Xavier initialization works well for Tanh networks. But what about ReLU networks? Here, we cannot take the previous assumption of the non-linearity becoming linear for small values. The ReLU activation function sets (in expectation) half of the inputs to 0 so that also the expectation of the input is not zero. However, as long as the expectation of W is zero and $b = 0$, the expectation of the output is zero. The part where the calculation of the ReLU initialization differs from the identity is when determining $\text{Var}(w_{ij}x_j)$:

$$\text{Var}(w_{ij}x_j) = \underbrace{\mathbb{E}[w_{ij}^2]}_{=\text{Var}(w_{ij})} \mathbb{E}[x_j^2] - \underbrace{\mathbb{E}[w_{ij}]^2}_{=0} \mathbb{E}[x_j]^2 = \text{Var}(w_{ij})\mathbb{E}[x_j^2]$$

If we assume now that x is the output of a ReLU activation (from a previous layer, $x = \max(0, \tilde{y})$), we can calculate the expectation as follows:

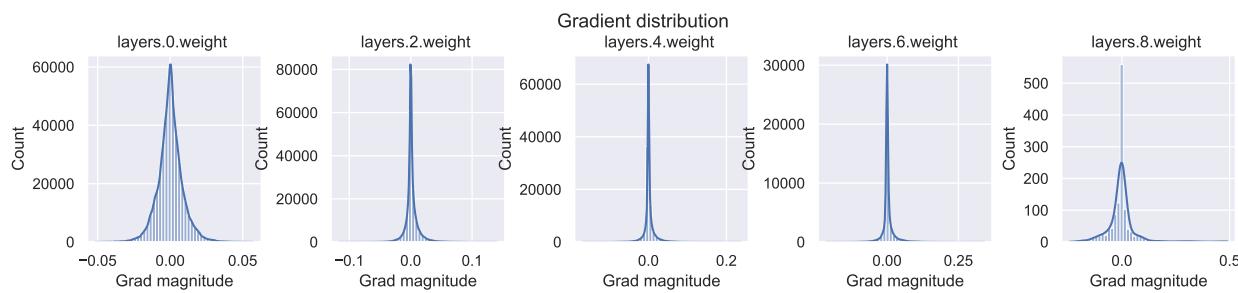
$$\begin{aligned} \mathbb{E}[x^2] &= \mathbb{E}[\max(0, \tilde{y})^2] \\ &= \frac{1}{2}\mathbb{E}[\tilde{y}^2] \quad \tilde{y} \text{ is zero-centered and symmetric} \\ &= \frac{1}{2}\text{Var}(\tilde{y}) \end{aligned}$$

Thus, we see that we have an additional factor of 1/2 in the equation, so that our desired weight variance becomes $2/d_x$. This gives us the Kaiming initialization (see He, K. et al. (2015)). Note that the Kaiming initialization does not use the harmonic mean between input and output size. In their paper (Section 2.2, Backward Propagation, last paragraph),

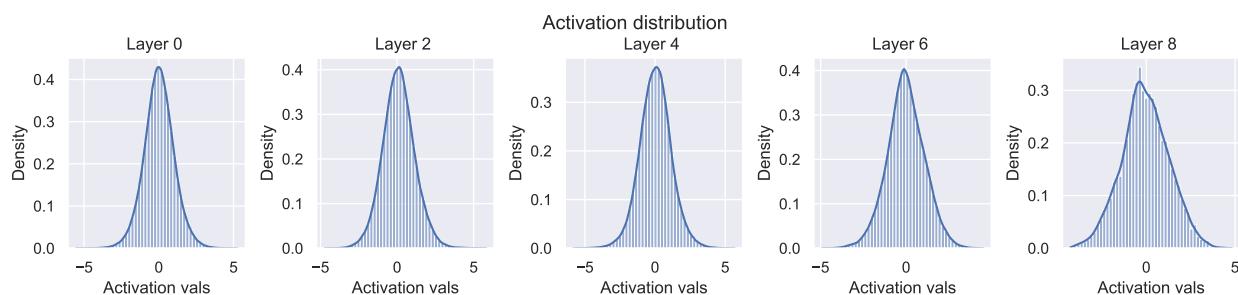
they argue that using d_x or d_y both lead to stable gradients throughout the network, and only depend on the overall input and output size of the network. Hence, we can use here only the input d_x :

```
[17]: def kaiming_init(model):
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            param.data.fill_(0)
        elif name.startswith("layers.0"): # The first layer does not have ReLU applied
            on its input
            param.data.normal_(0, 1/math.sqrt(param.shape[1]))
        else:
            param.data.normal_(0, math.sqrt(2)/math.sqrt(param.shape[1])))

model = BaseNetwork(act_fn=nn.ReLU()).to(device)
kaiming_init(model)
visualize_gradients(model, print_variance=True)
visualize_activations(model, print_variance=True)
```



```
layers.0.weight - Variance: 0.000075
layers.2.weight - Variance: 0.000108
layers.4.weight - Variance: 0.000185
layers.6.weight - Variance: 0.000444
layers.8.weight - Variance: 0.005548
```



```
Layer 0 - Variance: 1.012342
Layer 2 - Variance: 1.092432
Layer 4 - Variance: 1.268176
Layer 6 - Variance: 1.193706
Layer 8 - Variance: 1.760064
```

The variance stays stable across layers. We can conclude that the Kaiming initialization indeed works well for ReLU-based networks. Note that for Leaky-ReLU etc., we have to slightly adjust the factor of 2 in the variance as half of the values are not set to zero anymore. PyTorch provides a function to calculate this factor for many activation function, see `torch.nn.init.calculate_gain` ([link](#)).

4.7.3 Optimization

Besides initialization, selecting a suitable optimization algorithm can be an important choice for deep neural networks. Before taking a closer look at them, we should define code for training the models. Most of the following code is copied from the previous tutorial, and only slightly altered to fit our needs.

```
[18]: def _get_config_file(model_path, model_name):
    return os.path.join(model_path, model_name + ".config")

def _get_model_file(model_path, model_name):
    return os.path.join(model_path, model_name + ".tar")

def _get_result_file(model_path, model_name):
    return os.path.join(model_path, model_name + "_results.json")

def load_model(model_path, model_name, net=None):
    config_file, model_file = _get_config_file(model_path, model_name), _get_model_
    file(model_path, model_name)
    assert os.path.isfile(config_file), f"Could not find the config file \'{config_file}\'
    . Are you sure this is the correct path and you have your model config stored here?"
    assert os.path.isfile(model_file), f"Could not find the model file \'{model_file}\'.
    Are you sure this is the correct path and you have your model stored here?"
    with open(config_file, "r") as f:
        config_dict = json.load(f)
    if net is None:
        act_fn_name = config_dict["act_fn"].pop("name").lower()
        assert act_fn_name in act_fn_by_name, f"Unknown activation function \'{act_fn_
        name}\'. Please add it to the \'act_fn_by_name\' dict."
        act_fn = act_fn_by_name[act_fn_name]()
        net = BaseNetwork(act_fn=act_fn, **config_dict)
    net.load_state_dict(torch.load(model_file))
    return net

def save_model(model, model_path, model_name):
    config_dict = model.config
    os.makedirs(model_path, exist_ok=True)
    config_file, model_file = _get_config_file(model_path, model_name), _get_model_
    file(model_path, model_name)
    with open(config_file, "w") as f:
        json.dump(config_dict, f)
    torch.save(model.state_dict(), model_file)

def train_model(net, model_name, optim_func, max_epochs=50, batch_size=256,
    overwrite=False):
    """
    Train a model on the training set of FashionMNIST
    Inputs:
        net - Object of BaseNetwork
        model_name - (str) Name of the model, used for creating the checkpoint names
        max_epochs - Number of epochs we want to (maximally) train for
        patience - If the performance on the validation set has not improved for
        #patience epochs, we stop training early
    """

    Train a model on the training set of FashionMNIST
```

(continues on next page)

(continued from previous page)

```

batch_size - Size of batches used in training
overwrite - Determines how to handle the case when there already exists a
↪checkpoint. If True, it will be overwritten. Otherwise, we skip training.
"""

file_exists = os.path.isfile(_get_model_file(CHECKPOINT_PATH, model_name))
if file_exists and not overwrite:
    print(f"Model file of \'{model_name}\' already exists. Skipping training...")
    with open(_get_result_file(CHECKPOINT_PATH, model_name), "r") as f:
        results = json.load(f)
else:
    if file_exists:
        print("Model file exists, but will be overwritten...")

# Defining optimizer, loss and data loader
optimizer = optim_func(net.parameters())
loss_module = nn.CrossEntropyLoss()
train_loader_local = data.DataLoader(train_set, batch_size=batch_size,
↪shuffle=True, drop_last=True, pin_memory=True)

results = None
val_scores = []
train_losses, train_scores = [], []
best_val_epoch = -1
for epoch in range(max_epochs):
    #####
    # Training #
    #####
    net.train()
    true_preds, count = 0., 0
    t = tqdm(train_loader_local, leave=False)
    for imgs, labels in t:
        imgs, labels = imgs.to(device), labels.to(device)
        optimizer.zero_grad()
        preds = net(imgs)
        loss = loss_module(preds, labels)
        loss.backward()
        optimizer.step()
        # Record statistics during training
        true_preds += (preds.argmax(dim=-1) == labels).sum().item()
        count += labels.shape[0]
        t.set_description(f"Epoch {epoch+1}: loss={loss.item():4.2f}")
        train_losses.append(loss.item())
    train_acc = true_preds / count
    train_scores.append(train_acc)

    #####
    # Validation #
    #####
    val_acc = test_model(net, val_loader)
    val_scores.append(val_acc)
    print(f"[Epoch {epoch+1:2d}] Training accuracy: {train_acc*100.0:05.2f}%, "
↪Validation accuracy: {val_acc*100.0:05.2f}%")

```

(continues on next page)

(continued from previous page)

```

if len(val_scores) == 1 or val_acc > val_scores[best_val_epoch]:
    print("\t (New best performance, saving model...)")
    save_model(net, CHECKPOINT_PATH, model_name)
    best_val_epoch = epoch

if results is None:
    load_model(CHECKPOINT_PATH, model_name, net=net)
    test_acc = test_model(net, test_loader)
    results = {"test_acc": test_acc, "val_scores": val_scores, "train_losses": train_losses, "train_scores": train_scores}
    with open(_get_result_file(CHECKPOINT_PATH, model_name), "w") as f:
        json.dump(results, f)

# Plot a curve of the validation accuracy
sns.set()
plt.plot([i for i in range(1, len(results["train_scores"])+1)], results["train_scores"], label="Train")
plt.plot([i for i in range(1, len(results["val_scores"])+1)], results["val_scores"], label="Val")
plt.xlabel("Epochs")
plt.ylabel("Validation accuracy")
plt.ylim(min(results["val_scores"]), max(results["train_scores"])*1.01)
plt.title(f"Validation performance of {model_name}")
plt.legend()
plt.show()
plt.close()

print(f" Test accuracy: {results['test_acc']*100.0:4.2f}% ".center(50, "=")+"\n")
return results

def test_model(net, data_loader):
    """
    Test a model on a specified dataset.

    Inputs:
        net - Trained model of type BaseNetwork
        data_loader - DataLoader object of the dataset to test on (validation or test)
    """
    net.eval()
    true_preds, count = 0., 0
    for imgs, labels in data_loader:
        imgs, labels = imgs.to(device), labels.to(device)
        with torch.no_grad():
            preds = net(imgs).argmax(dim=-1)
            true_preds += (preds == labels).sum().item()
            count += labels.shape[0]
    test_acc = true_preds / count
    return test_acc

```

First, we need to understand what an optimizer actually does. The optimizer is responsible to update the network's parameters given the gradients. Hence, we effectively implement a function $w^t = f(w^{t-1}, g^t, \dots)$ with w being the

parameters, and $g^t = \nabla_{w^{(t-1)}} \mathcal{L}^{(t)}$ the gradients at time step t . A common, additional parameter to this function is the learning rate, here denoted by η . Usually, the learning rate can be seen as the “step size” of the update. A higher learning rate means that we change the weights more in the direction of the gradients, a smaller means we take shorter steps.

As most optimizers only differ in the implementation of f , we can define a template for an optimizer in PyTorch below. We take as input the parameters of a model and a learning rate. The function `zero_grad` sets the gradients of all parameters to zero, which we have to do before calling `loss.backward()`. Finally, the `step()` function tells the optimizer to update all weights based on their gradients. The template is setup below:

```
[19]: class OptimizerTemplate:

    def __init__(self, params, lr):
        self.params = list(params)
        self.lr = lr

    def zero_grad(self):
        ## Set gradients of all parameters to zero
        for p in self.params:
            if p.grad is not None:
                p.grad.detach_() # For second-order optimizers important
                p.grad.zero_()

    @torch.no_grad()
    def step(self):
        ## Apply update step to all parameters
        for p in self.params:
            if p.grad is None: # We skip parameters without any gradients
                continue
            self.update_param(p)

    def update_param(self, p):
        # To be implemented in optimizer-specific classes
        raise NotImplementedError
```

The first optimizer we are going to implement is the standard Stochastic Gradient Descent (SGD). SGD updates the parameters using the following equation:

$$w^{(t)} = w^{(t-1)} - \eta \cdot g^{(t)}$$

As simple as the equation is also our implementation of SGD:

```
[20]: class SGD(OptimizerTemplate):

    def __init__(self, params, lr):
        super().__init__(params, lr)

    def update_param(self, p):
        p_update = -self.lr * p.grad
        p.add_(p_update) # In-place update => saves memory and does not create
                         # computation graph
```

In the lecture, we also have discussed the concept of momentum which replaces the gradient in the update by an

exponential average of all past gradients including the current one:

$$\begin{aligned} m^{(t)} &= \beta_1 m^{(t-1)} + (1 - \beta_1) \cdot g^{(t)} \\ w^{(t)} &= w^{(t-1)} - \eta \cdot m^{(t)} \end{aligned}$$

Let's also implement it below:

```
[21]: class SGDMomentum(OptimizerTemplate):

    def __init__(self, params, lr, momentum=0.0):
        super().__init__(params, lr)
        self.momentum = momentum # Corresponds to beta_1 in the equation above
        self.param_momentum = {p: torch.zeros_like(p.data) for p in self.params} # Dict used
        ↵ to store m_t

    def update_param(self, p):
        self.param_momentum[p] = (1 - self.momentum) * p.grad + self.momentum * self.
        ↵ param_momentum[p]
        p_update = -self.lr * self.param_momentum[p]
        p.add_(p_update)
```

Finally, we arrive at Adam. Adam combines the idea of momentum with an adaptive learning rate, which is based on an exponential average of the squared gradients, i.e. the gradients norm. Furthermore, we add a bias correction for the momentum and adaptive learning rate for the first iterations:

$$\begin{aligned} m^{(t)} &= \beta_1 m^{(t-1)} + (1 - \beta_1) \cdot g^{(t)} \\ v^{(t)} &= \beta_2 v^{(t-1)} + (1 - \beta_2) \cdot (g^{(t)})^2 \\ \hat{m}^{(t)} &= \frac{m^{(t)}}{1 - \beta_1^t}, \hat{v}^{(t)} = \frac{v^{(t)}}{1 - \beta_2^t} \\ w^{(t)} &= w^{(t-1)} - \frac{\eta}{\sqrt{\hat{v}^{(t)}} + \epsilon} \circ \hat{m}^{(t)} \end{aligned}$$

Epsilon is a small constant used to improve numerical stability for very small gradient norms. Remember that the adaptive learning rate does not replace the learning rate hyperparameter η , but rather acts as an extra factor and ensures that the gradients of various parameters have a similar norm.

```
[22]: class Adam(OptimizerTemplate):

    def __init__(self, params, lr, beta1=0.9, beta2=0.999, eps=1e-8):
        super().__init__(params, lr)
        self.beta1 = beta1
        self.beta2 = beta2
        self.eps = eps
        self.param_step = {p: 0 for p in self.params} # Remembers "t" for each parameter
        ↵ for bias correction
        self.param_momentum = {p: torch.zeros_like(p.data) for p in self.params}
        self.param_2nd_momentum = {p: torch.zeros_like(p.data) for p in self.params}

    def update_param(self, p):
        self.param_step[p] += 1

        self.param_momentum[p] = (1 - self.beta1) * p.grad + self.beta1 * self.param_
        ↵ momentum[p]
```

(continues on next page)

(continued from previous page)

```

    self.param_2nd_momentum[p] = (1 - self.beta2) * (p.grad)**2 + self.beta2 * self.
    ↵param_2nd_momentum[p]

    bias_correction_1 = 1 - self.beta1 ** self.param_step[p]
    bias_correction_2 = 1 - self.beta2 ** self.param_step[p]

    p_2nd_mom = self.param_2nd_momentum[p] / bias_correction_2
    p_mom = self.param_momentum[p] / bias_correction_1
    p_lr = self.lr / (torch.sqrt(p_2nd_mom) + self.eps)
    p_update = -p_lr * p_mom

    p.add_(p_update)

```

Comparing optimizers on model training

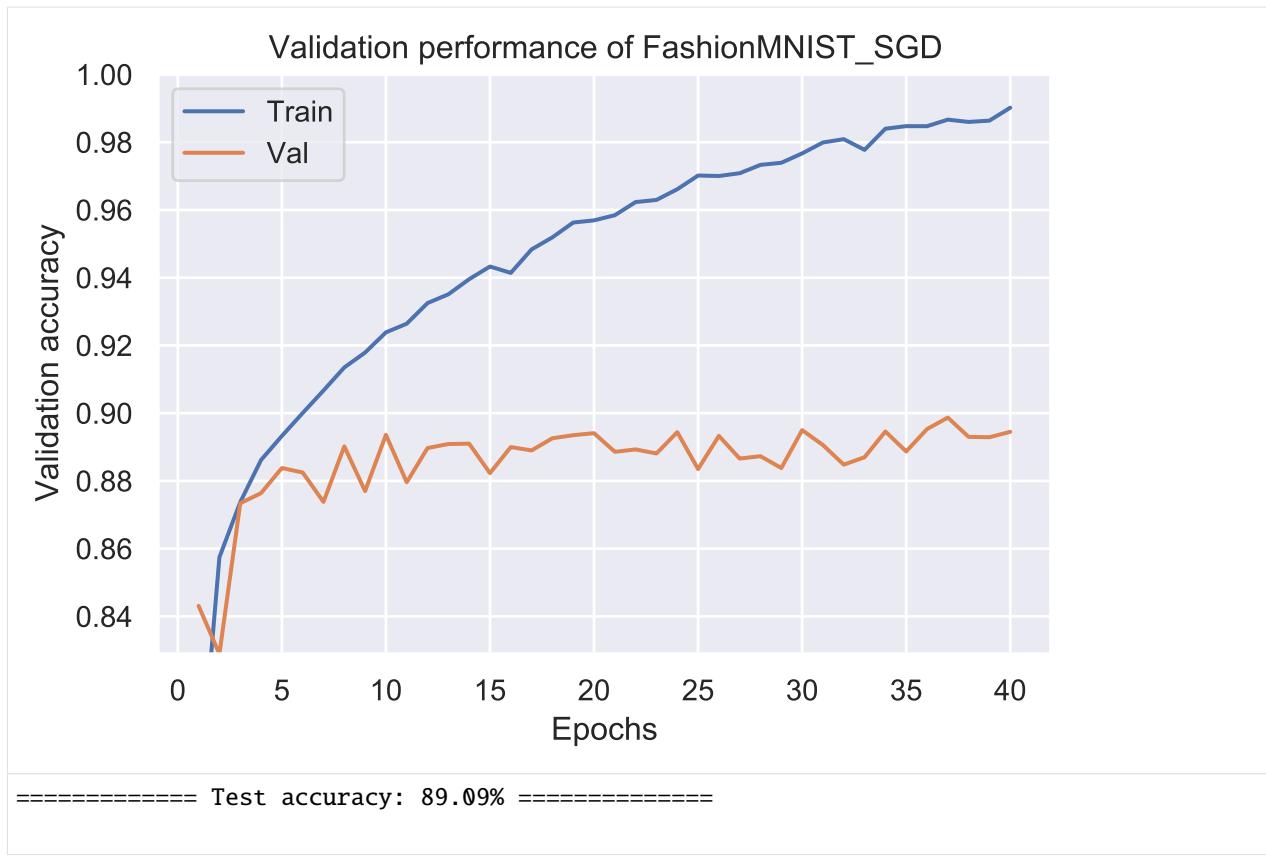
After we have implemented three optimizers (SGD, SGD with momentum, and Adam), we can start to analyze and compare them. First, we test them on how well they can optimize a neural network on the FashionMNIST dataset. We use again our linear network, this time with a ReLU activation and the kaiming initialization, which we have found before to work well for ReLU-based networks. Note that the model is over-parameterized for this task, and we can achieve similar performance with a much smaller network (for example 100, 100, 100). However, our main interest is in how well the optimizer can train *deep* neural networks, hence the over-parameterization.

```
[23]: base_model = BaseNetwork(act_fn=nn.ReLU(), hidden_sizes=[512, 256, 256, 128])
kaiming_init(base_model)
```

For a fair comparison, we train the exact same model with the same seed with the three optimizers below. Feel free to change the hyperparameters if you want (however, you have to train your own model then).

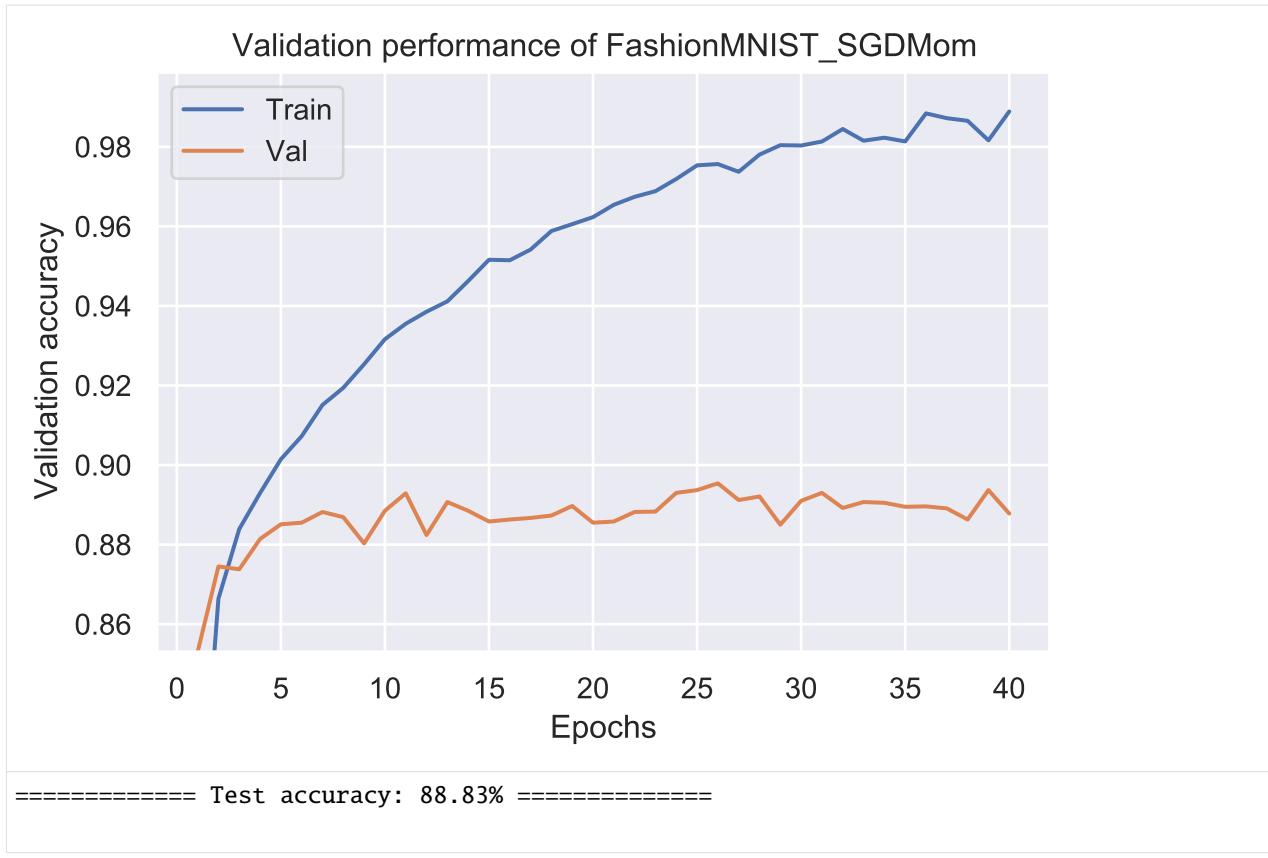
```
[24]: SGD_model = copy.deepcopy(base_model).to(device)
SGD_results = train_model(SGD_model, "FashionMNIST_SGD",
                           lambda params: SGD(params, lr=1e-1),
                           max_epochs=40, batch_size=256)

Model file of "FashionMNIST_SGD" already exists. Skipping training...
```



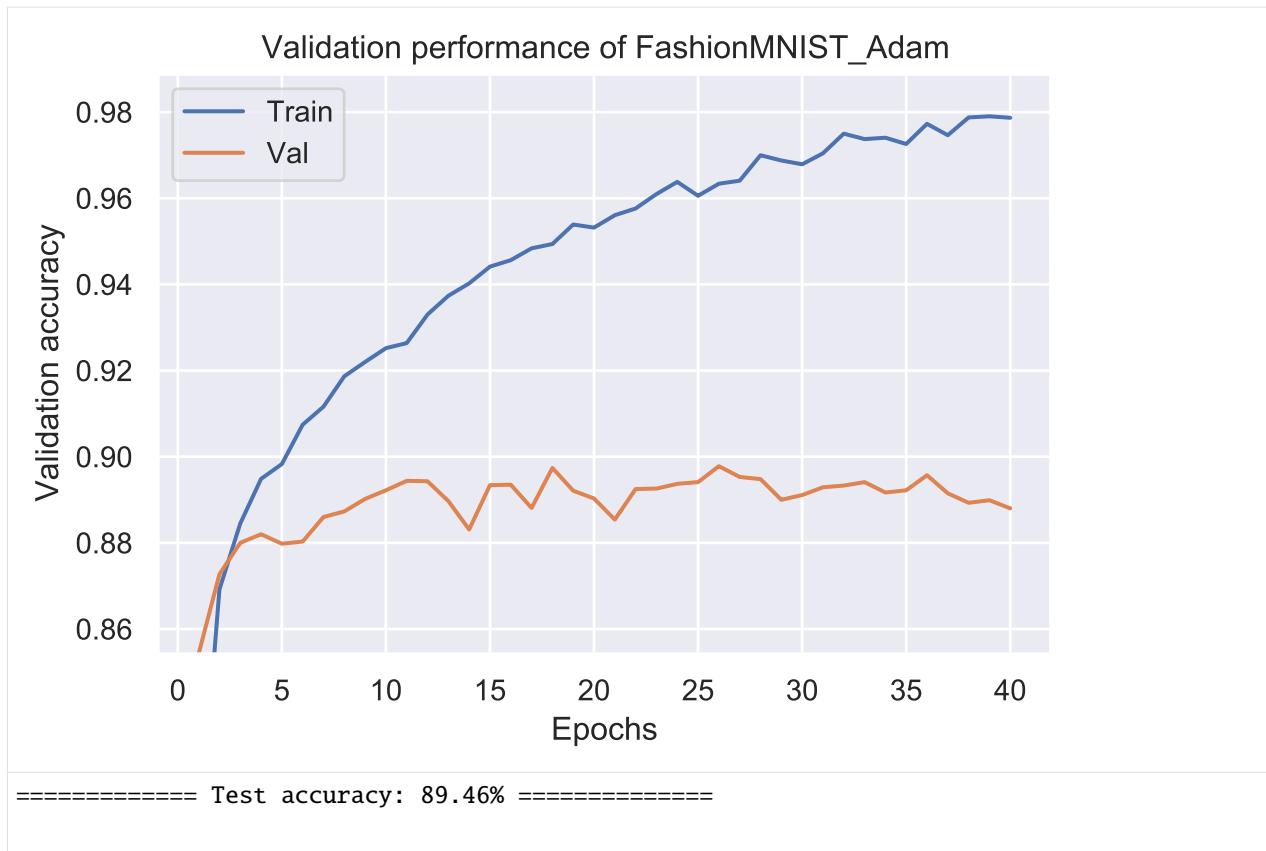
```
[25]: SGDMom_model = copy.deepcopy(base_model).to(device)
SGDMom_results = train_model(SGDMom_model, "FashionMNIST_SGDMom",
                               lambda params: SGDMomentum(params, lr=1e-1, momentum=0.9),
                               max_epochs=40, batch_size=256)

Model file of "FashionMNIST_SGDMom" already exists. Skipping training...
```



```
[26]: Adam_model = copy.deepcopy(base_model).to(device)
Adam_results = train_model(Adam_model, "FashionMNIST_Adam",
                           lambda params: Adam(params, lr=1e-3),
                           max_epochs=40, batch_size=256)
```

Model file of "FashionMNIST_Adam" already exists. Skipping training...



The result is that all optimizers perform similarly well with the given model. The differences are too small to find any significant conclusion. However, keep in mind that this can also be attributed to the initialization we chose. When changing the initialization to worse (e.g. constant initialization), Adam usually shows to be more robust because of its adaptive learning rate. To show the specific benefits of the optimizers, we will continue to look at some possible loss surfaces in which momentum and adaptive learning rate are crucial.

Pathological curvatures

A pathological curvature is a type of surface that is similar to ravines and is particularly tricky for plain SGD optimization. In words, pathological curvatures typically have a steep gradient in one direction with an optimum at the center, while in a second direction we have a slower gradient towards a (global) optimum. Let's first create an example surface of this and visualize it:

```
[27]: def pathological_curve_loss(w1, w2):
    # Example of a pathological curvature. There are many more possible, feel free to
    ↪experiment here!
    x1_loss = torch.tanh(w1)**2 + 0.01 * torch.abs(w1)
    x2_loss = torch.sigmoid(w2)
    return x1_loss + x2_loss
```

```
[28]: def plot_curve(curve_fn, x_range=(-5,5), y_range=(-5,5), plot_3d=False, cmap=cm.viridis, ↪
    title="Pathological curvature"):
    fig = plt.figure()
    ax = plt.axes(projection='3d') if plot_3d else plt.axes()
```

(continues on next page)

(continued from previous page)

```

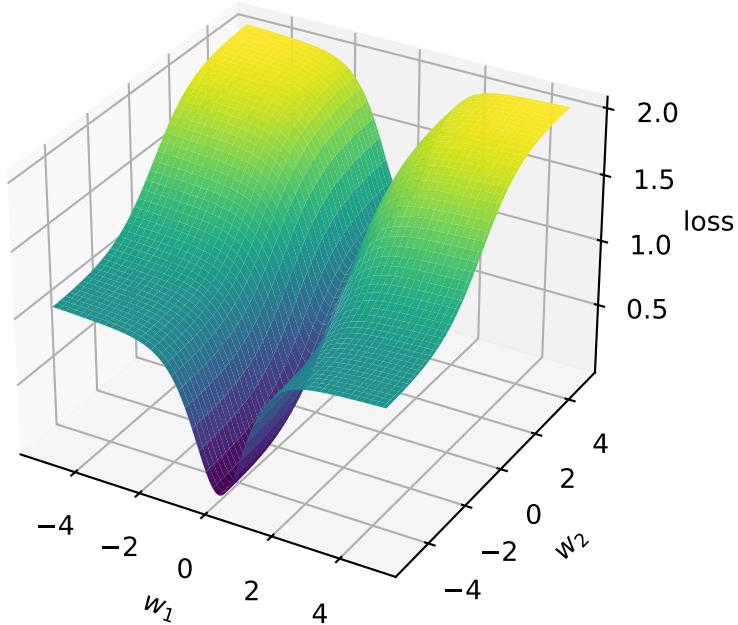
x = torch.arange(x_range[0], x_range[1], (x_range[1]-x_range[0])/100.)
y = torch.arange(y_range[0], y_range[1], (y_range[1]-y_range[0])/100.)
x, y = torch.meshgrid(x, y, indexing='xy')
z = curve_fn(x, y)
x, y, z = x.numpy(), y.numpy(), z.numpy()

if plot_3d:
    ax.plot_surface(x, y, z, cmap=cmap, linewidth=1, color="#000", antialiased=False)
    ax.set_zlabel("loss")
else:
    ax.imshow(z[::-1], cmap=cmap, extent=(x_range[0], x_range[1], y_range[0], y_range[1]))
    plt.title(title)
    ax.set_xlabel(r"$w_1$")
    ax.set_ylabel(r"$w_2$")
    plt.tight_layout()
return ax

sns.reset_orig()
_ = plot_curve(pathological_curve_loss, plot_3d=True)
plt.show()

```

Pathological curvature



In terms of optimization, you can image that w_1 and w_2 are weight parameters, and the curvature represents the loss surface over the space of w_1 and w_2 . Note that in typical networks, we have many, many more parameters than two, and such curvatures can occur in multi-dimensional spaces as well.

Ideally, our optimization algorithm would find the center of the ravine and focuses on optimizing the parameters towards the direction of w_2 . However, if we encounter a point along the ridges, the gradient is much greater in w_1 than w_2 , and we might end up jumping from one side to the other. Due to the large gradients, we would have to reduce our learning

rate slowing down learning significantly.

To test our algorithms, we can implement a simple function to train two parameters on such a surface:

```
[29]: def train_curve(optimizer_func, curve_func=pathological_curve_loss, num_updates=100, ↵
    ↵init=[5,5]): ↵
    """
    Inputs:
        optimizer_func - Constructor of the optimizer to use. Should only take a ↵
    ↵parameter list
        curve_func - Loss function (e.g. pathological curvature)
        num_updates - Number of updates/steps to take when optimizing
        init - Initial values of parameters. Must be a list/tuple with two elements, ↵
    ↵representing w_1 and w_2
    Outputs:
        Numpy array of shape [num_updates, 3] with [t,:2] being the parameter values at ↵
    ↵step t, and [t,2] the loss at t.
    """
    weights = nn.Parameter(torch.FloatTensor(init), requires_grad=True)
    optimizer = optimizer_func([weights])

    list_points = []
    for _ in range(num_updates):
        loss = curve_func(weights[0], weights[1])
        list_points.append(torch.cat([weights.data.detach(), loss.unsqueeze(dim=0).detach()], dim=0))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    points = torch.stack(list_points, dim=0).numpy()
    return points
```

Next, let's apply the different optimizers on our curvature. Note that we set a much higher learning rate for the optimization algorithms as you would in a standard neural network. This is because we only have 2 parameters instead of tens of thousands or even millions.

```
[30]: SGD_points = train_curve(lambda params: SGD(params, lr=10))
SGDMom_points = train_curve(lambda params: SGDMomentum(params, lr=10, momentum=0.9))
Adam_points = train_curve(lambda params: Adam(params, lr=1))
```

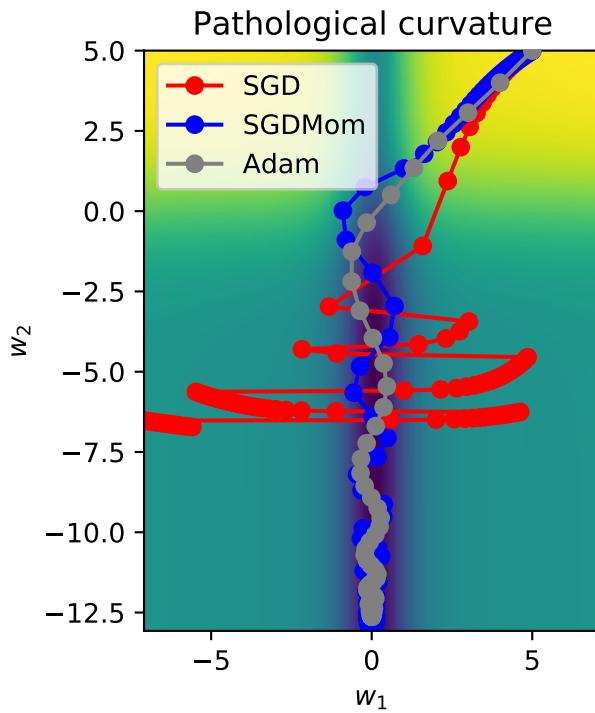
To understand best how the different algorithms worked, we visualize the update step as a line plot through the loss surface. We will stick with a 2D representation for readability.

```
[31]: all_points = np.concatenate([SGD_points, SGDMom_points, Adam_points], axis=0)
ax = plot_curve(pathological_curve_loss,
                 x_range=(-np.absolute(all_points[:,0]).max(), np.absolute(all_points[:,0]).max()),
                 y_range=(all_points[:,1].min(), all_points[:,1].max()),
                 plot_3d=False)
ax.plot(SGD_points[:,0], SGD_points[:,1], color="red", marker="o", zorder=1, label="SGD")
ax.plot(SGDMom_points[:,0], SGDMom_points[:,1], color="blue", marker="o", zorder=2, label="SGDMom")
ax.plot(Adam_points[:,0], Adam_points[:,1], color="grey", marker="o", zorder=3, label="Adam")
```

(continues on next page)

(continued from previous page)

```
plt.legend()
plt.show()
```



We can clearly see that SGD is not able to find the center of the optimization curve and has a problem converging due to the steep gradients in w_1 . In contrast, Adam and SGD with momentum nicely converge as the changing direction of w_1 is canceling itself out. On such surfaces, it is crucial to use momentum.

Steep optima

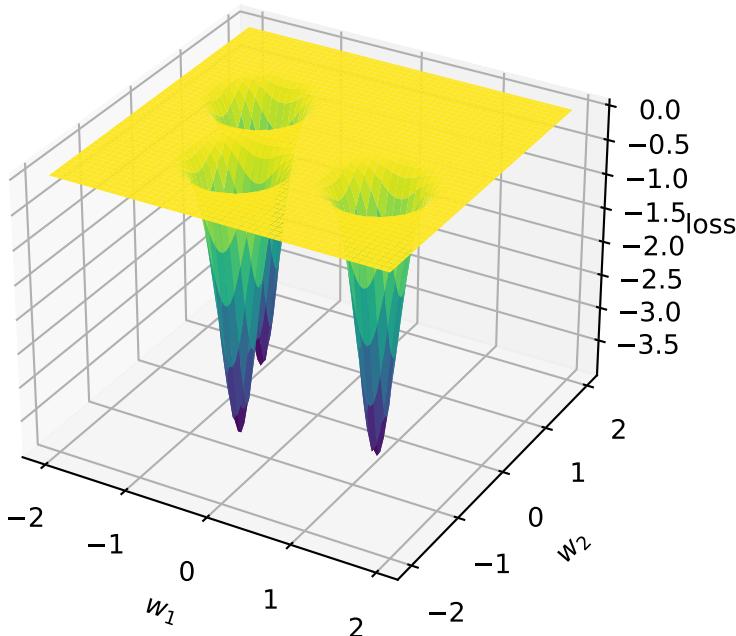
A second type of challenging loss surfaces are steep optima. In those, we have a larger part of the surface having very small gradients while around the optimum, we have very large gradients. For instance, take the following loss surfaces:

```
[32]: def bivar_gaussian(w1, w2, x_mean=0.0, y_mean=0.0, x_sig=1.0, y_sig=1.0):
    norm = 1 / (2 * np.pi * x_sig * y_sig)
    x_exp = (-1 * (w1 - x_mean)**2) / (2 * x_sig**2)
    y_exp = (-1 * (w2 - y_mean)**2) / (2 * y_sig**2)
    return norm * torch.exp(x_exp + y_exp)

def comb_func(w1, w2):
    z = -bivar_gaussian(w1, w2, x_mean=1.0, y_mean=-0.5, x_sig=0.2, y_sig=0.2)
    z -= bivar_gaussian(w1, w2, x_mean=-1.0, y_mean=0.5, x_sig=0.2, y_sig=0.2)
    z -= bivar_gaussian(w1, w2, x_mean=-0.5, y_mean=-0.8, x_sig=0.2, y_sig=0.2)
    return z

_ = plot_curve(comb_func, x_range=(-2, 2), y_range=(-2, 2), plot_3d=True, title="Steep ↴ optima")
```

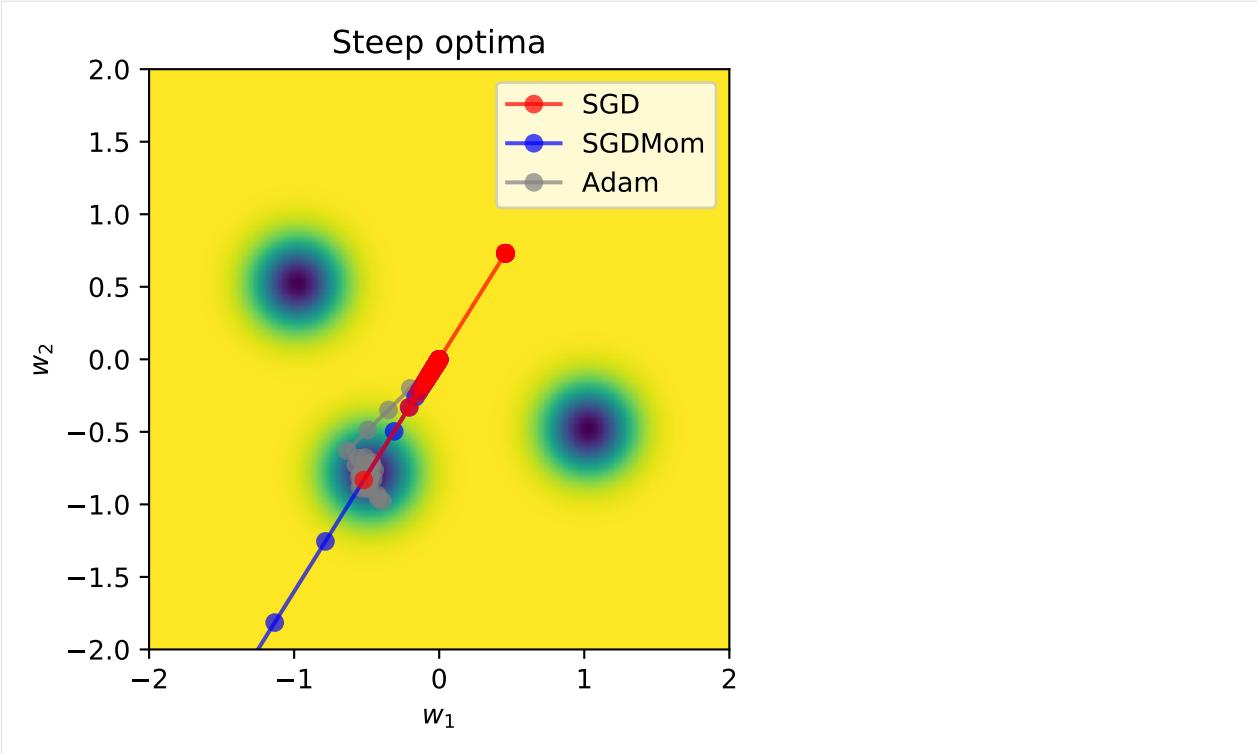
Steep optima



Most of the loss surface has very little to no gradients. However, close to the optima, we have very steep gradients. To reach the minimum when starting in a region with lower gradients, we expect an adaptive learning rate to be crucial. To verify this hypothesis, we can run our three optimizers on the surface:

```
[33]: SGD_points = train_curve(lambda params: SGD(params, lr=.5), comb_func, init=[0,0])
SGDMom_points = train_curve(lambda params: SGDMomentum(params, lr=1, momentum=0.9), comb_
    ↪ func, init=[0,0])
Adam_points = train_curve(lambda params: Adam(params, lr=0.2), comb_func, init=[0,0])

all_points = np.concatenate([SGD_points, SGDMom_points, Adam_points], axis=0)
ax = plot_curve(comb_func,
                 x_range=(-2, 2),
                 y_range=(-2, 2),
                 plot_3d=False,
                 title="Steep optima")
ax.plot(SGD_points[:,0], SGD_points[:,1], color="red", marker="o", zorder=3, label="SGD",
    ↪ alpha=0.7)
ax.plot(SGDMom_points[:,0], SGDMom_points[:,1], color="blue", marker="o", zorder=2, ↪
    ↪ label="SGDMom", alpha=0.7)
ax.plot(Adam_points[:,0], Adam_points[:,1], color="grey", marker="o", zorder=1, label=
    ↪ "Adam", alpha=0.7)
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
plt.legend()
plt.show()
```



SGD first takes very small steps until it touches the border of the optimum. First reaching a point around $(-0.75, -0.5)$, the gradient direction has changed and pushes the parameters to $(0.8, 0.5)$ from which SGD cannot recover anymore (only with many, many steps). A similar problem has SGD with momentum, only that it continues the direction of the touch of the optimum. The gradients from this time step are so much larger than any other point that the momentum m_t is overpowered by it. Finally, Adam is able to converge in the optimum showing the importance of adaptive learning rates.

What optimizer to take

After seeing the results on optimization, what is our conclusion? Should we always use Adam and never look at SGD anymore? The short answer: no. There are many papers saying that in certain situations, SGD (with momentum) generalizes better where Adam often tends to overfit [5,6]. This is related to the idea of finding wider optima. For instance, see the illustration of different optima below (credit: [Keskar et al., 2017](#)):

The black line represents the training loss surface, while the dotted red line is the test loss. Finding sharp, narrow minima can be helpful for finding the minimal training loss. However, this doesn't mean that it also minimizes the test loss as especially flat minima have shown to generalize better. You can imagine that the test dataset has a slightly shifted loss surface due to the different examples than in the training set. A small change can have a significant influence for sharp minima, while flat minima are generally more robust to this change.

In the next tutorial, we will see that some network types can still be better optimized with SGD and learning rate scheduling than Adam. Nevertheless, Adam is the most commonly used optimizer in Deep Learning as it usually performs better than other optimizers, especially for deep networks.

4.7.4 Conclusion

In this tutorial, we have looked at initialization and optimization techniques for neural networks. We have seen that a good initialization has to balance the preservation of the gradient variance as well as the activation variance. This can be achieved with the Xavier initialization for tanh-based networks, and the Kaiming initialization for ReLU-based networks. In optimization, concepts like momentum and adaptive learning rate can help with challenging loss surfaces but don't guarantee an increase in performance for neural networks.

4.7.5 References

- [1] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010. [link](#)
- [2] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE international conference on computer vision. 2015. [link](#)
- [3] Kingma, Diederik P. & Ba, Jimmy. "Adam: A Method for Stochastic Optimization." Proceedings of the third international conference for learning representations (ICLR). 2015. [link](#)
- [4] Keskar, Nitish Shirish, et al. "On large-batch training for deep learning: Generalization gap and sharp minima." Proceedings of the fifth international conference for learning representations (ICLR). 2017. [link](#)
- [5] Wilson, Ashia C., et al. "The Marginal Value of Adaptive Gradient Methods in Machine Learning." Advances in neural information processing systems. 2017. [link](#)
- [6] Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint. 2017. [link](#)

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.8 Tutorial 5: Inception, ResNet and DenseNet

Filled notebook:

Pre-trained models:

Recordings:

JAX+Flax version:

Author: Phillip Lippe

Note: Interested in JAX? Check out our [JAX+Flax version](#) of this tutorial!

In this tutorial, we will implement and discuss variants of modern CNN architectures. There have been many different architectures been proposed over the past few years. Some of the most impactful ones, and still relevant today, are

the following: GoogleNet/Inception architecture (winner of ILSVRC 2014), ResNet (winner of ILSVRC 2015), and DenseNet (best paper award CVPR 2017). All of them were state-of-the-art models when being proposed, and the core ideas of these networks are the foundations for most current state-of-the-art architectures. Thus, it is important to understand these architectures in detail and learn how to implement them.

Let's start with importing our standard libraries here.

```
[1]: ## Standard libraries
import os
import numpy as np
import random
from PIL import Image
from types import SimpleNamespace

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## PyTorch
import torch
import torch.nn as nn
import torch.utils.data as data
import torch.optim as optim
# Torchvision
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms
```

We will use the same `set_seed` function as in the previous tutorials, as well as the path variables `DATASET_PATH` and `CHECKPOINT_PATH`. Adjust the paths if necessary.

```
[2]: # Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial5"

# Function for setting the seed
def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
set_seed(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
```

(continues on next page)

(continued from previous page)

```
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
```

We also have pretrained models and Tensorboards (more on this later) for this tutorial, and download them below.

```
[3]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorials5/"
# Files to download
pretrained_files = ["GoogleNet.ckpt", "ResNet.ckpt", "ResNetPreAct.ckpt", "DenseNet.ckpt",
                    "tensorboards/GoogleNet/events.out.tfevents.googlenet",
                    "tensorboards/ResNet/events.out.tfevents.resnet",
                    "tensorboards/ResNetPreAct/events.out.tfevents.resnetpreact",
                    "tensorboards/DenseNet/events.out.tfevents.densenet"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive folder, or contact the author with the full output including the following error:\n", e)
```

Throughout this tutorial, we will train and evaluate the models on the CIFAR10 dataset. This allows you to compare the results obtained here with the model you have implemented in the first assignment. As we have learned from the previous tutorial about initialization, it is important to have the data preprocessed with a zero mean. Therefore, as a first step, we will calculate the mean and standard deviation of the CIFAR dataset:

```
[4]: train_dataset = CIFAR10(root=DATASET_PATH, train=True, download=True)
DATA_MEANS = (train_dataset.data / 255.0).mean(axis=(0,1,2))
DATA_STD = (train_dataset.data / 255.0).std(axis=(0,1,2))
print("Data mean", DATA_MEANS)
print("Data std", DATA_STD)

Files already downloaded and verified
Data mean [0.49139968 0.48215841 0.44653091]
Data std [0.24703223 0.24348513 0.26158784]
```

We will use this information to define a `transforms.Normalize` module which will normalize our data accordingly. Additionally, we will use data augmentation during training. This reduces the risk of overfitting and helps CNNs to generalize better. Specifically, we will apply two random augmentations.

First, we will flip each image horizontally by a chance of 50% (`transforms.RandomHorizontalFlip`). The object class usually does not change when flipping an image, and we don't expect any image information to be dependent on the horizontal orientation. This would be however different if we would try to detect digits or letters in an image, as those have a certain orientation.

The second augmentation we use is called `transforms.RandomResizedCrop`. This transformation scales the image in a small range, while eventually changing the aspect ratio, and crops it afterward in the previous size. Therefore, the actual pixel values change while the content or overall semantics of the image stays the same.

We will randomly split the training dataset into a training and a validation set. The validation set will be used for determining early stopping. After finishing the training, we test the models on the CIFAR test set.

```
[5]: test_transform = transforms.Compose([transforms.ToTensor(),
                                         transforms.Normalize(DATA_MEANS, DATA_STD)
                                         ])
# For training, we add some augmentation. Networks are too powerful and would overfit.
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                         transforms.RandomResizedCrop((32,32),scale=(0.8,1.
                                         ↵0),ratio=(0.9,1.1)),
                                         transforms.ToTensor(),
                                         transforms.Normalize(DATA_MEANS, DATA_STD)
                                         ])
# Loading the training dataset. We need to split it into a training and validation part
# We need to do a little trick because the validation set should not use the
# ↵augmentation.
train_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=train_transform,
                         ↵download=True)
val_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=test_transform,
                         ↵download=True)
set_seed(42)
train_set, _ = torch.utils.data.random_split(train_dataset, [45000, 5000])
set_seed(42)
_, val_set = torch.utils.data.random_split(val_dataset, [45000, 5000])

# Loading the test set
test_set = CIFAR10(root=DATASET_PATH, train=False, transform=test_transform,
                         ↵download=True)

# We define a set of data loaders that we can use for various purposes later.
train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True,
                         ↵pin_memory=True, num_workers=4)
val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False, drop_last=False,
                         ↵num_workers=4)
test_loader = data.DataLoader(test_set, batch_size=128, shuffle=False, drop_last=False,
                         ↵num_workers=4)

Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

To verify that our normalization works, we can print out the mean and standard deviation of the single batch. The mean should be close to 0 and the standard deviation close to 1 for each channel:

```
[6]: imgs, _ = next(iter(train_loader))
print("Batch mean", imgs.mean(dim=[0,2,3]))
```

(continues on next page)

(continued from previous page)

```
print("Batch std", imgs.std(dim=[0,2,3]))
Batch mean tensor([-0.0143, -0.0204, -0.0017])
Batch std tensor([0.9772, 0.9650, 0.9809])
```

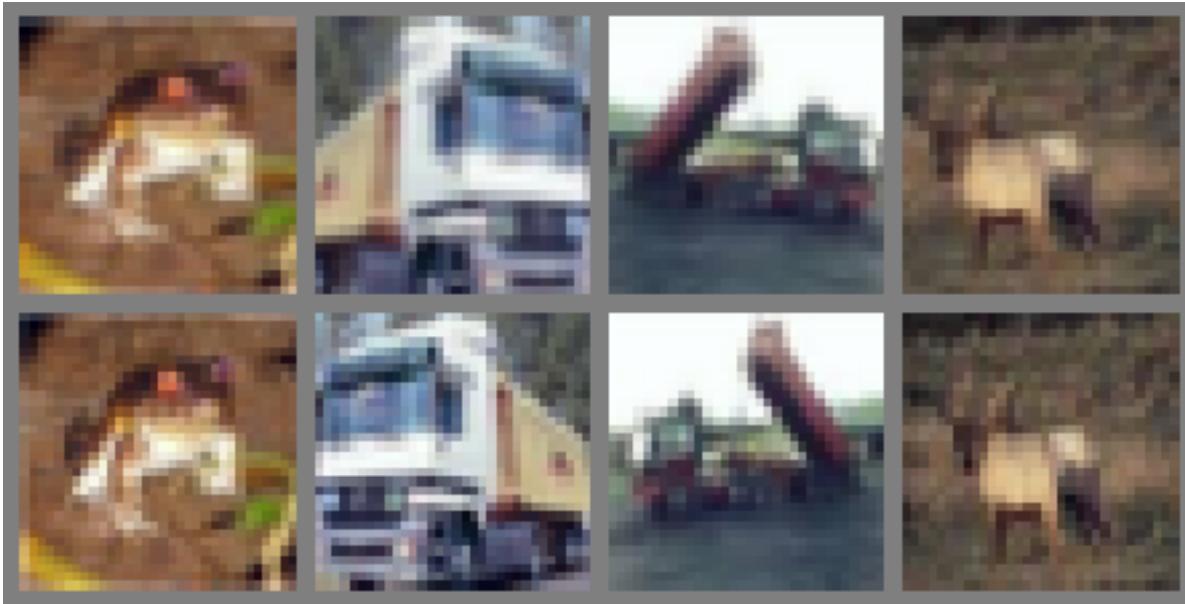
Finally, let's visualize a few images from the training set, and how they look like after random data augmentation:

```
[7]: NUM_IMAGES = 4
images = [train_dataset[idx] for idx in range(NUM_IMAGES)]
orig_images = [Image.fromarray(train_dataset.data[idx]) for idx in range(NUM_IMAGES)]
orig_images = [test_transform(img) for img in orig_images]

img_grid = torchvision.utils.make_grid(torch.stack(images + orig_images, dim=0), nrow=4, normalize=True, pad_value=0.5)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("Augmentation examples on CIFAR10")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

Augmentation examples on CIFAR10



4.8.1 PyTorch Lightning

In this notebook and in many following ones, we will make use of the library [PyTorch Lightning](#). PyTorch Lightning is a framework that simplifies your code needed to train, evaluate, and test a model in PyTorch. It also handles logging into [TensorBoard](#), a visualization toolkit for ML experiments, and saving model checkpoints automatically with minimal code overhead from our side. This is extremely helpful for us as we want to focus on implementing different model architectures and spend little time on other code overhead. Note that at the time of writing/teaching, the framework has been released in version 1.6. Future versions might have a slightly changed interface and thus might not work perfectly with the code (we will try to keep it up-to-date as much as possible).

Now, we will take the first step in PyTorch Lightning, and continue to explore the framework in our other tutorials. First, we import the library:

```
[8]: # PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.5
    import pytorch_lightning as pl
```

PyTorch Lightning comes with a lot of useful functions, such as one for setting the seed:

```
[9]: # Setting the seed
pl.seed_everything(42)
```

```
[9]: 42
```

Thus, in the future, we don't have to define our own `set_seed` function anymore.

In PyTorch Lightning, we define `pl.LightningModule`'s (inheriting from `torch.nn.Module`) that organize our code into 5 main sections:

1. Initialization (`__init__`), where we create all necessary parameters/models
2. Optimizers (`configure_optimizers`) where we create the optimizers, learning rate scheduler, etc.
3. Training loop (`training_step`) where we only have to define the loss calculation for a single batch (the loop of `optimizer.zero_grad()`, `loss.backward()` and `optimizer.step()`, as well as any logging/saving operation, is done in the background)
4. Validation loop (`validation_step`) where similarly to the training, we only have to define what should happen per step
5. Test loop (`test_step`) which is the same as validation, only on a test set.

Therefore, we don't abstract the PyTorch code, but rather organize it and define some default operations that are commonly used. If you need to change something else in your training/validation/test loop, there are many possible functions you can overwrite (see the [docs](#) for details).

Now we can look at an example of how a Lightning Module for training a CNN looks like:

```
[10]: class CIFARModule(pl.LightningModule):
    def __init__(self, model_name, model_hparams, optimizer_name, optimizer_hparams):
        """
        Inputs:
            model_name - Name of the model/CNN to run. Used for creating the model (see function below)
        """

```

(continues on next page)

(continued from previous page)

```

model_hparams - Hyperparameters for the model, as dictionary.
optimizer_name - Name of the optimizer to use. Currently supported: Adam, SGD
optimizer_hparams - Hyperparameters for the optimizer, as dictionary. This ↵
↳ includes learning rate, weight decay, etc.
"""

super().__init__()
# Exports the hyperparameters to a YAML file, and create "self.hparams" namespace
self.save_hyperparameters()
# Create model
self.model = create_model(model_name, model_hparams)
# Create loss module
self.loss_module = nn.CrossEntropyLoss()
# Example input for visualizing the graph in Tensorboard
self.example_input_array = torch.zeros((1, 3, 32, 32), dtype=torch.float32)

def forward(self, imgs):
    # Forward function that is run when visualizing the graph
    return self.model(imgs)

def configure_optimizers(self):
    # We will support Adam or SGD as optimizers.
    if self.hparams.optimizer_name == "Adam":
        # AdamW is Adam with a correct implementation of weight decay (see here for ↵
        ↳ details: https://arxiv.org/pdf/1711.05101.pdf)
        optimizer = optim.AdamW(
            self.parameters(), **self.hparams.optimizer_hparams)
    elif self.hparams.optimizer_name == "SGD":
        optimizer = optim.SGD(self.parameters(), **self.hparams.optimizer_hparams)
    else:
        assert False, f"Unknown optimizer: \'{self.hparams.optimizer_name}\'"

    # We will reduce the learning rate by 0.1 after 100 and 150 epochs
    scheduler = optim.lr_scheduler.MultiStepLR(
        optimizer, milestones=[100, 150], gamma=0.1)
    return [optimizer], [scheduler]

def training_step(self, batch, batch_idx):
    # "batch" is the output of the training data loader.
    imgs, labels = batch
    preds = self.model(imgs)
    loss = self.loss_module(preds, labels)
    acc = (preds.argmax(dim=-1) == labels).float().mean()

    # Logs the accuracy per epoch to tensorboard (weighted average over batches)
    self.log('train_acc', acc, on_step=False, on_epoch=True)
    self.log('train_loss', loss)
    return loss # Return tensor to call ".backward" on

def validation_step(self, batch, batch_idx):
    imgs, labels = batch
    preds = self.model(imgs).argmax(dim=-1)
    acc = (labels == preds).float().mean()

```

(continues on next page)

(continued from previous page)

```
# By default logs it per epoch (weighted average over batches)
self.log('val_acc', acc)

def test_step(self, batch, batch_idx):
    imgs, labels = batch
    preds = self.model(imgs).argmax(dim=-1)
    acc = (labels == preds).float().mean()
    # By default logs it per epoch (weighted average over batches), and returns it afterwards
    self.log('test_acc', acc)
```

We see that the code is organized and clear, which helps if someone else tries to understand your code.

Another important part of PyTorch Lightning is the concept of callbacks. Callbacks are self-contained functions that contain the non-essential logic of your Lightning Module. They are usually called after finishing a training epoch, but can also influence other parts of your training loop. For instance, we will use the following two pre-defined callbacks: `LearningRateMonitor` and `ModelCheckpoint`. The learning rate monitor adds the current learning rate to our TensorBoard, which helps to verify that our learning rate scheduler works correctly. The model checkpoint callback allows you to customize the saving routine of your checkpoints. For instance, how many checkpoints to keep, when to save, which metric to look out for, etc. We import them below:

```
[11]: # Callbacks
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
```

To allow running multiple different models with the same Lightning module, we define a function below that maps a model name to the model class. At this stage, the dictionary `model_dict` is empty, but we will fill it throughout the notebook with our new models.

```
[12]: model_dict = {}

def create_model(model_name, model_hparams):
    if model_name in model_dict:
        return model_dict[model_name](**model_hparams)
    else:
        assert False, f"Unknown model name \'{model_name}\'. Available models are:
→ {str(model_dict.keys())}"
```

Similarly, to use the activation function as another hyperparameter in our model, we define a “name to function” dict below:

```
[13]: act_fn_by_name = {
    "tanh": nn.Tanh,
    "relu": nn.ReLU,
    "leakyrelu": nn.LeakyReLU,
    "gelu": nn.GELU
}
```

If we pass the classes or objects directly as an argument to the Lightning module, we couldn’t take advantage of PyTorch Lightning’s automatically hyperparameter saving and loading.

Besides the Lightning module, the second most important module in PyTorch Lightning is the `Trainer`. The trainer is responsible to execute the training steps defined in the Lightning module and completes the framework. Similar to the Lightning module, you can override any key part that you don’t want to be automated, but the default settings are often the best practice to do. For a full overview, see the [documentation](#). The most important functions we use below are:

- `trainer.fit`: Takes as input a lightning module, a training dataset, and an (optional) validation dataset. This function trains the given module on the training dataset with occasional validation (default once per epoch, can be changed)
- `trainer.test`: Takes as input a model and a dataset on which we want to test. It returns the test metric on the dataset.

For training and testing, we don't have to worry about things like setting the model to eval mode (`model.eval()`) as this is all done automatically. See below how we define a training function for our models:

```
[14]: def train_model(model_name, save_name=None, **kwargs):
    """
    Inputs:
        model_name - Name of the model you want to run. Is used to look up the class in
        "model_dict"
        save_name (optional) - If specified, this name will be used for creating the
        checkpoint and logging directory.
    """

    if save_name is None:
        save_name = model_name

    # Create a PyTorch Lightning trainer with the generation callback
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, save_name),
    # Where to save models
    accelerator="gpu" if str(device).startswith("cuda") else "cpu",
    # We run on a GPU (if possible)
    devices=1,
    # How many GPUs/CPUs we want to use (1 is enough for the notebooks)
    max_epochs=180,
    # How many epochs to train for if no patience is set
    callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
    monitor="val_acc"), # Save the best checkpoint based on the maximum val_acc recorded.
    # Saves only weights and not optimizer
    LearningRateMonitor("epoch")],
    # Log learning rate every epoch
    enable_progress_bar=True)
    # Set to False if you do not want a progress bar
    trainer.logger._log_graph = True # If True, we plot the computation graph in
    tensorboard
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't
    need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, save_name + ".ckpt")
    if os.path.isfile(pretrained_filename):
        print(f"Found pretrained model at {pretrained_filename}, loading...")
        model = CIFARModule.load_from_checkpoint(pretrained_filename) # Automatically
    loads the model with the saved hyperparameters
    else:
        pl.seed_everything(42) # To be reproducible
        model = CIFARModule(model_name=model_name, **kwargs)
        trainer.fit(model, train_loader, val_loader)
        model = CIFARModule.load_from_checkpoint(trainer.checkpoint_callback.best_model_
    path) # Load best checkpoint after training
```

(continues on next page)

(continued from previous page)

```
# Test best model on validation and test set
val_result = trainer.test(model, val_loader, verbose=False)
test_result = trainer.test(model, test_loader, verbose=False)
result = {"test": test_result[0]["test_acc"], "val": val_result[0]["test_acc"]}

return model, result
```

Finally, we can focus on the Convolutional Neural Networks we want to implement today: GoogleNet, ResNet, and DenseNet.

4.8.2 Inception

The [GoogleNet](#), proposed in 2014, won the ImageNet Challenge because of its usage of the Inception modules. In general, we will mainly focus on the concept of Inception in this tutorial instead of the specifics of the GoogleNet, as based on Inception, there have been many follow-up works ([Inception-v2](#), [Inception-v3](#), [Inception-v4](#), [Inception-ResNet](#),...). The follow-up works mainly focus on increasing efficiency and enabling very deep Inception networks. However, for a fundamental understanding, it is sufficient to look at the original Inception block.

An Inception block applies four convolution blocks separately on the same feature map: a 1x1, 3x3, and 5x5 convolution, and a max pool operation. This allows the network to look at the same data with different receptive fields. Of course, learning only 5x5 convolution would be theoretically more powerful. However, this is not only more computation and memory heavy but also tends to overfit much easier. The overall inception block looks like below (figure credit - [Szegedy et al.](#)):

The additional 1x1 convolutions before the 3x3 and 5x5 convolutions are used for dimensionality reduction. This is especially crucial as the feature maps of all branches are merged afterward, and we don't want any explosion of feature size. As 5x5 convolutions are 25 times more expensive than 1x1 convolutions, we can save a lot of computation and parameters by reducing the dimensionality before the large convolutions.

We can now try to implement the Inception Block ourselves:

```
[15]: class InceptionBlock(nn.Module):

    def __init__(self, c_in, c_red : dict, c_out : dict, act_fn):
        """
        Inputs:
            c_in - Number of input feature maps from the previous layers
            c_red - Dictionary with keys "3x3" and "5x5" specifying the output of the
                    dimensionality reducing 1x1 convolutions
            c_out - Dictionary with keys "1x1", "3x3", "5x5", and "max"
            act_fn - Activation class constructor (e.g. nn.ReLU)
        """
        super().__init__()

        # 1x1 convolution branch
        self.conv_1x1 = nn.Sequential(
            nn.Conv2d(c_in, c_out["1x1"], kernel_size=1),
            nn.BatchNorm2d(c_out["1x1"]),
            act_fn()
        )
```

(continues on next page)

(continued from previous page)

```

# 3x3 convolution branch
self.conv_3x3 = nn.Sequential(
    nn.Conv2d(c_in, c_red["3x3"], kernel_size=1),
    nn.BatchNorm2d(c_red["3x3"]),
    act_fn(),
    nn.Conv2d(c_red["3x3"], c_out["3x3"], kernel_size=3, padding=1),
    nn.BatchNorm2d(c_out["3x3"]),
    act_fn()
)

# 5x5 convolution branch
self.conv_5x5 = nn.Sequential(
    nn.Conv2d(c_in, c_red["5x5"], kernel_size=1),
    nn.BatchNorm2d(c_red["5x5"]),
    act_fn(),
    nn.Conv2d(c_red["5x5"], c_out["5x5"], kernel_size=5, padding=2),
    nn.BatchNorm2d(c_out["5x5"]),
    act_fn()
)

# Max-pool branch
self.max_pool = nn.Sequential(
    nn.MaxPool2d(kernel_size=3, padding=1, stride=1),
    nn.Conv2d(c_in, c_out["max"], kernel_size=1),
    nn.BatchNorm2d(c_out["max"]),
    act_fn()
)

def forward(self, x):
    x_1x1 = self.conv_1x1(x)
    x_3x3 = self.conv_3x3(x)
    x_5x5 = self.conv_5x5(x)
    x_max = self.max_pool(x)
    x_out = torch.cat([x_1x1, x_3x3, x_5x5, x_max], dim=1)
    return x_out

```

The GoogleNet architecture consists of stacking multiple Inception blocks with occasional max pooling to reduce the height and width of the feature maps. The original GoogleNet was designed for image sizes of ImageNet (224x224 pixels) and had almost 7 million parameters. As we train on CIFAR10 with image sizes of 32x32, we don't require such a heavy architecture, and instead, apply a reduced version. The number of channels for dimensionality reduction and output per filter (1x1, 3x3, 5x5, and max pooling) need to be manually specified and can be changed if interested. The general intuition is to have the most filters for the 3x3 convolutions, as they are powerful enough to take the context into account while requiring almost a third of the parameters of the 5x5 convolution.

```
[16]: class GoogleNet(nn.Module):

    def __init__(self, num_classes=10, act_fn_name="relu", **kwargs):
        super().__init__()
        self.hparams = SimpleNamespace(num_classes=num_classes,
                                      act_fn_name=act_fn_name,
                                      act_fn=act_fn_by_name[act_fn_name])
```

(continues on next page)

(continued from previous page)

```

self._create_network()
self._init_params()

def _create_network(self):
    # A first convolution on the original image to scale up the channel size
    self.input_net = nn.Sequential(
        nn.Conv2d(3, 64, kernel_size=3, padding=1),
        nn.BatchNorm2d(64),
        self.hparams.act_fn()
    )
    # Stacking inception blocks
    self.inception_blocks = nn.Sequential(
        InceptionBlock(64, c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 16, "3x3": 32,
        "5x5": 8, "max": 8}, act_fn=self.hparams.act_fn),
        InceptionBlock(64, c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 24, "3x3": 48,
        "5x5": 12, "max": 12}, act_fn=self.hparams.act_fn),
        nn.MaxPool2d(3, stride=2, padding=1), # 32x32 => 16x16
        InceptionBlock(96, c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 24, "3x3": 48,
        "5x5": 12, "max": 12}, act_fn=self.hparams.act_fn),
        InceptionBlock(96, c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 16, "3x3": 48,
        "5x5": 16, "max": 16}, act_fn=self.hparams.act_fn),
        InceptionBlock(96, c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 16, "3x3": 48,
        "5x5": 16, "max": 16}, act_fn=self.hparams.act_fn),
        InceptionBlock(96, c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 32, "3x3": 48,
        "5x5": 24, "max": 24}, act_fn=self.hparams.act_fn),
        nn.MaxPool2d(3, stride=2, padding=1), # 16x16 => 8x8
        InceptionBlock(128, c_red={"3x3": 48, "5x5": 16}, c_out={"1x1": 32, "3x3": 64,
        "5x5": 16, "max": 16}, act_fn=self.hparams.act_fn),
        InceptionBlock(128, c_red={"3x3": 48, "5x5": 16}, c_out={"1x1": 32, "3x3": 64,
        "5x5": 16, "max": 16}, act_fn=self.hparams.act_fn)
    )
    # Mapping to classification output
    self.output_net = nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)),
        nn.Flatten(),
        nn.Linear(128, self.hparams.num_classes)
    )
)

def _init_params(self):
    # Based on our discussion in Tutorial 4, we should initialize the convolutions
    # according to the activation function
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(
                m.weight, nonlinearity=self.hparams.act_fn_name)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)

def forward(self, x):
    x = self.input_net(x)
    x = self.inception_blocks(x)

```

(continues on next page)

(continued from previous page)

```
x = self.output_net(x)
return x
```

Now, we can integrate our model to the model dictionary we defined above:

```
[17]: model_dict["GoogleNet"] = GoogleNet
```

The training of the model is handled by PyTorch Lightning, and we just have to define the command to start. Note that we train for almost 200 epochs, which takes about an hour on Lisa's default GPUs (GTX1080Ti). We would recommend using the saved models and train your own model if you are interested.

```
[18]: googlenet_model, googlenet_results = train_model(model_name="GoogleNet",
                                                     model_hparams={"num_classes": 10,
                                                                    "act_fn_name": "relu"},
                                                     optimizer_name="Adam",
                                                     optimizer_hparams={"lr": 1e-3,
                                                                        "weight_decay": 1e-4})
→
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Found pretrained model at ../saved_models/tutorial5/GoogleNet.ckpt, loading...
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',
→ layout=Layout(flex='2')), max=...)
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',
→ layout=Layout(flex='2')), max=...))
```

We will compare the results later in the notebooks, but we can already print them here for a first glance:

```
[19]: print("GoogleNet Results", googlenet_results)
```

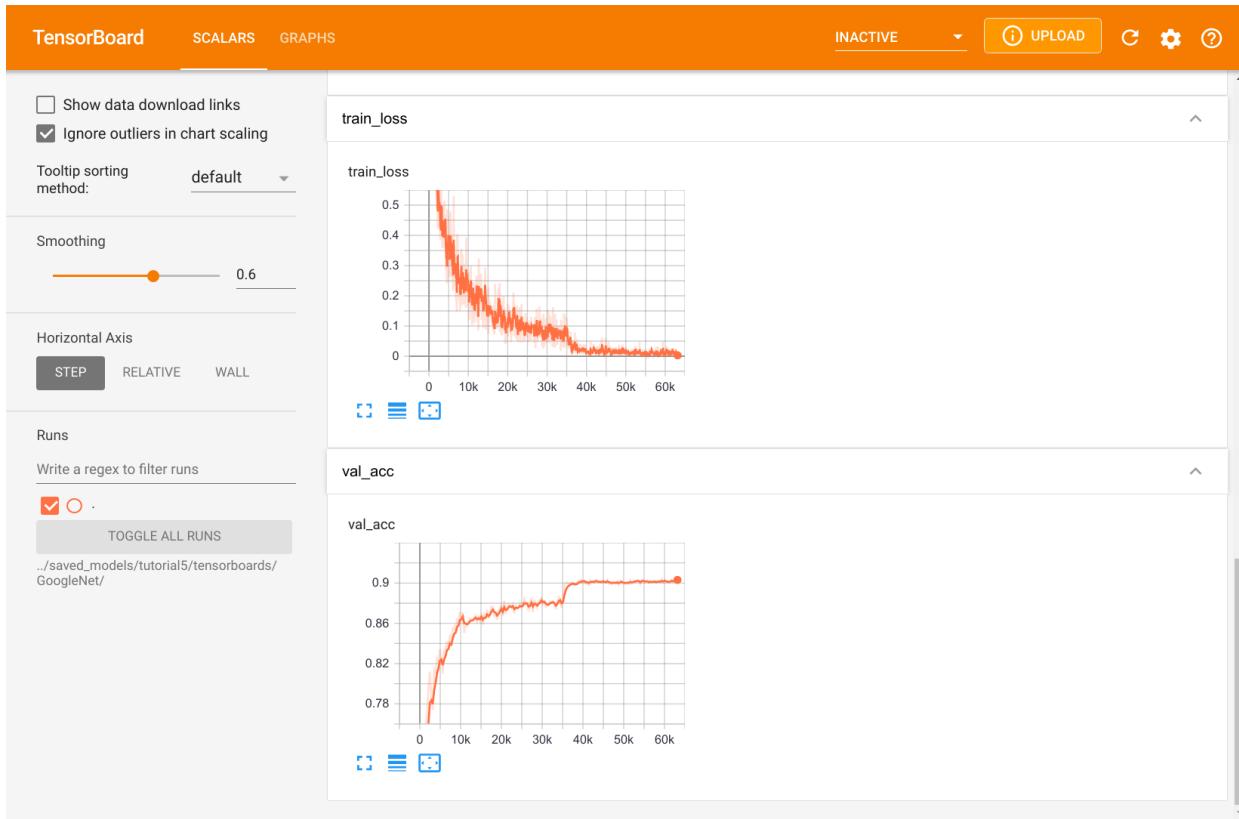
```
GoogleNet Results {'test': 0.8970000147819519, 'val': 0.9039999842643738}
```

Tensorboard log

A nice extra of PyTorch Lightning is the automatic logging into TensorBoard. To give you a better intuition of what TensorBoard can be used, we can look at the board that PyTorch Lightning has been generated when training the GoogleNet. TensorBoard provides an inline functionality for Jupyter notebooks, and we use it here:

```
[20]: # Load tensorboard extension
%load_ext tensorboard
```

```
[21]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH!
%tensorboard --logdir ../saved_models/tutorial5/tensorboards/GoogleNet/
```



TensorBoard is organized in multiple tabs. The main tab is the scalar tab where we can log the development of single numbers. For example, we have plotted the training loss, accuracy, learning rate, etc. If we look at the training or validation accuracy, we can really see the impact of using a learning rate scheduler. Reducing the learning rate gives our model a nice increase in training performance. Similarly, when looking at the training loss, we see a sudden decrease at this point. However, the high numbers on the training set compared to validation indicate that our model was overfitting which is inevitable for such large networks.

Another interesting tab in TensorBoard is the graph tab. It shows us the network architecture organized by building blocks from the input to the output. It basically shows the operations taken in the forward step of `CIFARModule`. Double-click on a module to open it. Feel free to explore the architecture from a different perspective. The graph visualization can often help you to validate that your model is actually doing what it is supposed to do, and you don't miss any layers in the computation graph.

4.8.3 ResNet

The [ResNet](#) paper is one of the [most cited AI papers](#), and has been the foundation for neural networks with more than 1,000 layers. Despite its simplicity, the idea of residual connections is highly effective as it supports stable gradient propagation through the network. Instead of modeling $x_{l+1} = F(x_l)$, we model $x_{l+1} = x_l + F(x_l)$ where F is a non-linear mapping (usually a sequence of NN modules like convolutions, activation functions, and normalizations). If we do backpropagation on such residual connections, we obtain:

$$\frac{\partial x_{l+1}}{\partial x_l} = \mathbf{I} + \frac{\partial F(x_l)}{\partial x_l}$$

The bias towards the identity matrix guarantees a stable gradient propagation being less effected by F itself. There have been many variants of ResNet proposed, which mostly concern the function F , or operations applied on the sum. In this tutorial, we look at two of them: the original ResNet block, and the [Pre-Activation ResNet block](#). We visually compare the blocks below (figure credit - [He et al.](#)):

The original ResNet block applies a non-linear activation function, usually ReLU, after the skip connection. In contrast, the pre-activation ResNet block applies the non-linearity at the beginning of F . Both have their advantages and disadvantages. For very deep network, however, the pre-activation ResNet has shown to perform better as the gradient flow is guaranteed to have the identity matrix as calculated above, and is not harmed by any non-linear activation applied to it. For comparison, in this notebook, we implement both ResNet types as shallow networks.

Let's start with the original ResNet block. The visualization above already shows what layers are included in F . One special case we have to handle is when we want to reduce the image dimensions in terms of width and height. The basic ResNet block requires $F(x_l)$ to be of the same shape as x_l . Thus, we need to change the dimensionality of x_l as well before adding to $F(x_l)$. The original implementation used an identity mapping with stride 2 and padded additional feature dimensions with 0. However, the more common implementation is to use a 1x1 convolution with stride 2 as it allows us to change the feature dimensionality while being efficient in parameter and computation cost. The code for the ResNet block is relatively simple, and shown below:

```
[22]: class ResNetBlock(nn.Module):

    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        """
        Inputs:
            c_in - Number of input features
            act_fn - Activation class constructor (e.g. nn.ReLU)
            subsample - If True, we want to apply a stride inside the block and reduce
            ↪ the output shape by 2 in height and width
            c_out - Number of output features. Note that this is only relevant if
            ↪ subsample is True, as otherwise, c_out = c_in
        """
        super().__init__()
        if not subsample:
            c_out = c_in

        # Network representing F
        self.net = nn.Sequential(
            nn.Conv2d(c_in, c_out, kernel_size=3, padding=1, stride=1 if not subsample
            ↪ else 2, bias=False), # No bias needed as the Batch Norm handles it
            nn.BatchNorm2d(c_out),
            act_fn(),
            nn.Conv2d(c_out, c_out, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(c_out)
        )

        # 1x1 convolution with stride 2 means we take the upper left value, and
        ↪ transform it to new output size
        self.downsample = nn.Conv2d(c_in, c_out, kernel_size=1, stride=2) if subsample
        ↪ else None
        self.act_fn = act_fn()

    def forward(self, x):
        z = self.net(x)
        if self.downsample is not None:
            x = self.downsample(x)
        out = z + x
        out = self.act_fn(out)
        return out
```

The second block we implement is the pre-activation ResNet block. For this, we have to change the order of layer in `self.net`, and do not apply an activation function on the output. Additionally, the downsampling operation has to apply a non-linearity as well as the input, x_l , has not been processed by a non-linearity yet. Hence, the block looks as follows:

```
[23]: class PreActResNetBlock(nn.Module):

    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        """
        Inputs:
            c_in - Number of input features
            act_fn - Activation class constructor (e.g. nn.ReLU)
            subsample - If True, we want to apply a stride inside the block and reduce
                        the output shape by 2 in height and width
            c_out - Number of output features. Note that this is only relevant if
                    subsample is True, as otherwise, c_out = c_in
        """
        super().__init__()
        if not subsample:
            c_out = c_in

        # Network representing F
        self.net = nn.Sequential(
            nn.BatchNorm2d(c_in),
            act_fn(),
            nn.Conv2d(c_in, c_out, kernel_size=3, padding=1, stride=1 if not subsample
                     else 2, bias=False),
            nn.BatchNorm2d(c_out),
            act_fn(),
            nn.Conv2d(c_out, c_out, kernel_size=3, padding=1, bias=False)
        )

        # 1x1 convolution needs to apply non-linearity as well as not done on skip
        # connection
        self.downsample = nn.Sequential(
            nn.BatchNorm2d(c_in),
            act_fn(),
            nn.Conv2d(c_in, c_out, kernel_size=1, stride=2, bias=False)
        ) if subsample else None

    def forward(self, x):
        z = self.net(x)
        if self.downsample is not None:
            x = self.downsample(x)
        out = z + x
        return out
```

Similarly to the model selection, we define a dictionary to create a mapping from string to block class. We will use the string name as hyperparameter value in our model to choose between the ResNet blocks. Feel free to implement any other ResNet block type and add it here as well.

```
[24]: resnet_blocks_by_name = {
    "ResNetBlock": ResNetBlock,
    "PreActResNetBlock": PreActResNetBlock
```

(continues on next page)

(continued from previous page)

}

The overall ResNet architecture consists of stacking multiple ResNet blocks, of which some are downsampling the input. When talking about ResNet blocks in the whole network, we usually group them by the same output shape. Hence, if we say the ResNet has [3, 3, 3] blocks, it means that we have 3 times a group of 3 ResNet blocks, where a subsampling is taking place in the fourth and seventh block. The ResNet with [3, 3, 3] blocks on CIFAR10 is visualized below.

The three groups operate on the resolutions 32×32 , 16×16 and 8×8 respectively. The blocks in orange denote ResNet blocks with downsampling. The same notation is used by many other implementations such as in the `torchvision` library from PyTorch. Thus, our code looks as follows:

```
[25]: class ResNet(nn.Module):

    def __init__(self, num_classes=10, num_blocks=[3,3,3], c_hidden=[16,32,64], act_fn_name="relu", block_name="ResNetBlock", **kwargs):
        """
        Inputs:
            num_classes - Number of classification outputs (10 for CIFAR10)
            num_blocks - List with the number of ResNet blocks to use. The first block of each group uses downsampling, except the first.
            c_hidden - List with the hidden dimensionalities in the different blocks. Usually multiplied by 2 the deeper we go.
            act_fn_name - Name of the activation function to use, looked up in "act_fn_by_name"
            block_name - Name of the ResNet block, looked up in "resnet_blocks_by_name"
        """
        super().__init__()
        assert block_name in resnet_blocks_by_name
        self.hparams = SimpleNamespace(num_classes=num_classes,
                                       c_hidden=c_hidden,
                                       num_blocks=num_blocks,
                                       act_fn_name=act_fn_name,
                                       act_fn=act_fn_by_name[act_fn_name],
                                       block_class=resnet_blocks_by_name[block_name])

        self._create_network()
        self._init_params()

    def _create_network(self):
        c_hidden = self.hparams.c_hidden

        # A first convolution on the original image to scale up the channel size
        if self.hparams.block_class == PreActResNetBlock: # => Don't apply non-linearity on output
            self.input_net = nn.Sequential(
                nn.Conv2d(3, c_hidden[0], kernel_size=3, padding=1, bias=False)
            )
        else:
            self.input_net = nn.Sequential(
                nn.Conv2d(3, c_hidden[0], kernel_size=3, padding=1, bias=False),
                nn.BatchNorm2d(c_hidden[0]),
                self.hparams.act_fn()
            )


```

(continues on next page)

(continued from previous page)

```

        )

    # Creating the ResNet blocks
blocks = []
for block_idx, block_count in enumerate(self.hparams.num_blocks):
    for bc in range(block_count):
        subsample = (bc == 0 and block_idx > 0) # Subsample the first block of
        ↪each group, except the very first one.
        blocks.append(
            self.hparams.block_class(c_in=c_hidden[block_idx] if not subsample
        ↪else (block_idx-1)],
            act_fn=self.hparams.act_fn,
            subsample=subsample,
            c_out=c_hidden[block_idx])
    )
self.blocks = nn.Sequential(*blocks)

# Mapping to classification output
self.output_net = nn.Sequential(
    nn.AdaptiveAvgPool2d((1,1)),
    nn.Flatten(),
    nn.Linear(c_hidden[-1], self.hparams.num_classes)
)

def _init_params(self):
    # Based on our discussion in Tutorial 4, we should initialize the convolutions
    ↪according to the activation function
    # Fan-out focuses on the gradient distribution, and is commonly used in ResNets
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity=self.
        ↪hparams.act_fn_name)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)

def forward(self, x):
    x = self.input_net(x)
    x = self.blocks(x)
    x = self.output_net(x)
    return x

```

We also need to add the new ResNet class to our model dictionary:

[26]: model_dict["ResNet"] = ResNet

Finally, we can train our ResNet models. One difference to the GoogleNet training is that we explicitly use SGD with Momentum as optimizer instead of Adam. Adam often leads to a slightly worse accuracy on plain, shallow ResNets. It is not 100% clear why Adam performs worse in this context, but one possible explanation is related to ResNet's loss surface. ResNet has been shown to produce smoother loss surfaces than networks without skip connection (see Li et al., 2018 for details). A possible visualization of the loss surface with/out skip connections is below (figure credit - Li et al.):

The x and y axis shows a projection of the parameter space, and the z axis shows the loss values achieved by different parameter values. On smooth surfaces like the one on the right, we might not require an adaptive learning rate as Adam provides. Instead, Adam can get stuck in local optima while SGD finds the wider minima that tend to generalize better. However, to answer this question in detail, we would need an extra tutorial because it is not easy to answer. For now, we conclude: for ResNet architectures, consider the optimizer to be an important hyperparameter, and try training with both Adam and SGD. Let's train the model below with SGD:

```
[27]: resnet_model, resnet_results = train_model(model_name="ResNet",
                                                model_hparams={"num_classes": 10,
                                                               "c_hidden": [16, 32, 64],
                                                               "num_blocks": [3, 3, 3],
                                                               "act_fn_name": "relu"},
                                                optimizer_name="SGD",
                                                optimizer_hparams={"lr": 0.1,
                                                               "momentum": 0.9,
                                                               "weight_decay": 1e-4})

GPU available: True, used: True
WARNING: Logging before flag parsing goes to stderr.
I1109 15:33:03.301807 139927971620672 distributed.py:49] GPU available: True, used: True
TPU available: False, using: 0 TPU cores
I1109 15:33:03.303210 139927971620672 distributed.py:49] TPU available: False, using: 0 ↵
TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
I1109 15:33:03.304439 139927971620672 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_ ↵
VISIBLE_DEVICES: [0]

Found pretrained model at ../saved_models/tutorial5/ResNet.ckpt, loading...
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', ↵
layout=Layout(flex='2')), max=...))

HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', ↵
layout=Layout(flex='2')), max=...))
```

Let's also train the pre-activation ResNet as comparison:

```
[28]: resnetpreact_model, resnetpreact_results = train_model(model_name="ResNet",
                                                       model_hparams={"num_classes": 10,
                                                               "c_hidden": [16, 32, 64],
                                                               "num_blocks": [3, 3],
                                                               "act_fn_name": "relu",
                                                               "block_name": "PreActResNetBlock"},

                                                       optimizer_name="SGD",
                                                       optimizer_hparams={"lr": 0.1,
                                                               "momentum": 0.9,}
```

(continues on next page)

(continued from previous page)

```
        "weight_decay":  
    ↵ 1e-4},  
    save_name="ResNetPreAct")
```

```
GPU available: True, used: True  
I1109 15:33:06.813361 139927971620672 distributed.py:49] GPU available: True, used: True  
TPU available: False, using: 0 TPU cores  
I1109 15:33:06.815118 139927971620672 distributed.py:49] TPU available: False, using: 0 ↵  
    ↵TPU cores  
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]  
I1109 15:33:06.816660 139927971620672 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_ ↵  
    ↵VISIBLE_DEVICES: [0]
```

```
Found pretrained model at ../saved_models/tutorial5/ResNetPreAct.ckpt, loading...
```

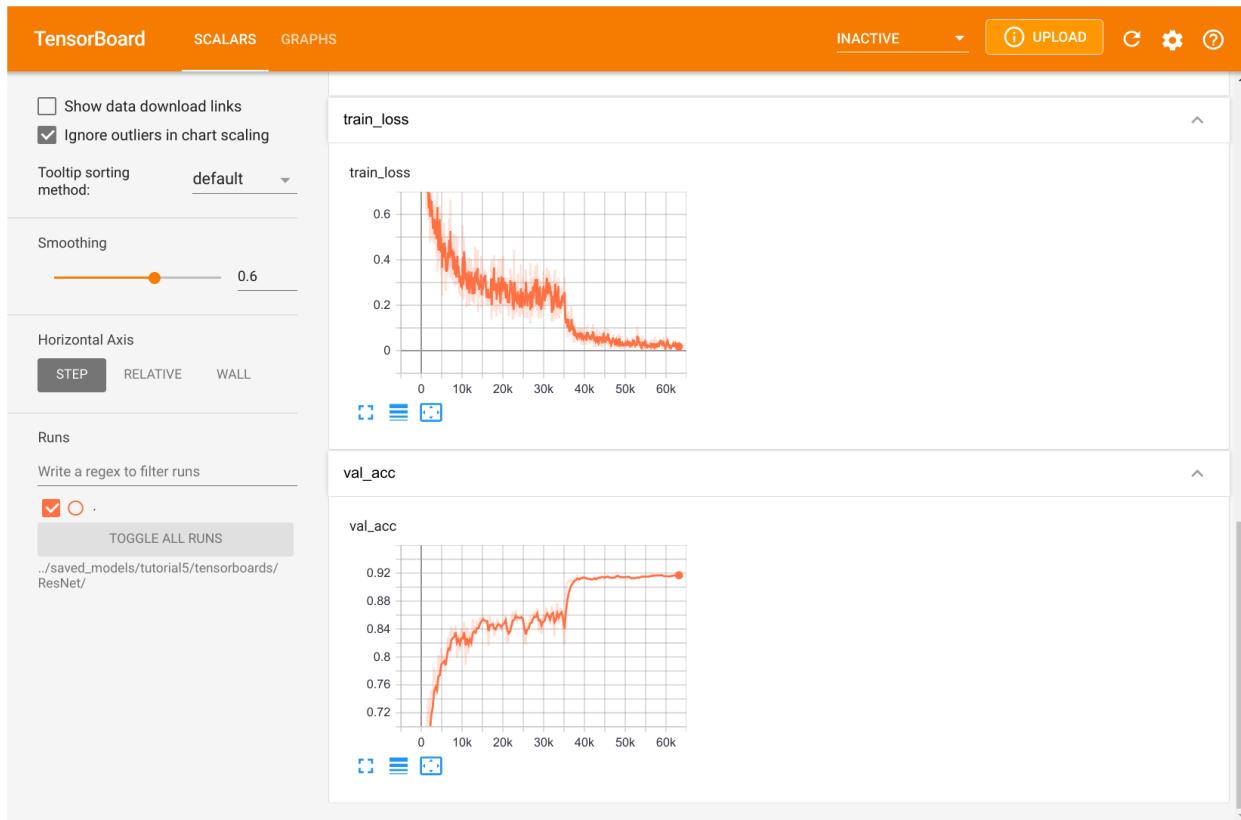
```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', ↵  
    ↵layout=Layout(flex='2')), max=...)
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', ↵  
    ↵layout=Layout(flex='2')), max=...)
```

Tensorboard log

Similarly to our GoogleNet model, we also have a TensorBoard log for the ResNet model. We can open it below.

```
[29]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH! Feel free to ↵  
    ↵change "ResNet" to "ResNetPreAct"  
%tensorboard --logdir ../saved_models/tutorial5/tensorboards/ResNet/
```



Feel free to explore the TensorBoard yourself, including the computation graph. In general, we can see that with SGD, the ResNet has a higher training loss than the GoogleNet in the first stage of the training. After reducing the learning rate however, the model achieves even higher validation accuracies. We compare the precise scores at the end of the notebook.

4.8.4 DenseNet

DenseNet is another architecture for enabling very deep neural networks and takes a slightly different perspective on residual connections. Instead of modeling the difference between layers, DenseNet considers residual connections as a possible way to reuse features across layers, removing any necessity to learn redundant feature maps. If we go deeper into the network, the model learns abstract features to recognize patterns. However, some complex patterns consist of a combination of abstract features (e.g. hand, face, etc.), and low-level features (e.g. edges, basic color, etc.). To find these low-level features in the deep layers, standard CNNs have to learn copy such feature maps, which wastes a lot of parameter complexity. DenseNet provides an efficient way of reusing features by having each convolution depends on all previous input features, but add only a small amount of filters to it. See the figure below for an illustration (figure credit - [Hu et al.](#)):

The last layer, called the transition layer, is responsible for reducing the dimensionality of the feature maps in height, width, and channel size. Although those technically break the identity backpropagation, there are only a few in a network so that it doesn't affect the gradient flow much.

We split the implementation of the layers in DenseNet into three parts: a `DenseLayer`, and a `DenseBlock`, and a `TransitionLayer`. The module `DenseLayer` implements a single layer inside a dense block. It applies a 1×1 convolution for dimensionality reduction with a subsequent 3×3 convolution. The output channels are concatenated to the originals and returned. Note that we apply the Batch Normalization as the first layer of each block. This allows slightly

different activations for the same features to different layers, depending on what is needed. Overall, we can implement it as follows:

```
[30]: class DenseLayer(nn.Module):

    def __init__(self, c_in, bn_size, growth_rate, act_fn):
        """
        Inputs:
            c_in - Number of input channels
            bn_size - Bottleneck size (factor of growth rate) for the output of the 1x1 convolution. Typically between 2 and 4.
            growth_rate - Number of output channels of the 3x3 convolution
            act_fn - Activation class constructor (e.g. nn.ReLU)
        """
        super().__init__()
        self.net = nn.Sequential(
            nn.BatchNorm2d(c_in),
            act_fn(),
            nn.Conv2d(c_in, bn_size * growth_rate, kernel_size=1, bias=False),
            nn.BatchNorm2d(bn_size * growth_rate),
            act_fn(),
            nn.Conv2d(bn_size * growth_rate, growth_rate, kernel_size=3, padding=1, bias=False)
        )

    def forward(self, x):
        out = self.net(x)
        out = torch.cat([out, x], dim=1)
        return out
```

The module `DenseBlock` summarizes multiple dense layers applied in sequence. Each dense layer takes as input the original input concatenated with all previous layers' feature maps:

```
[31]: class DenseBlock(nn.Module):

    def __init__(self, c_in, num_layers, bn_size, growth_rate, act_fn):
        """
        Inputs:
            c_in - Number of input channels
            num_layers - Number of dense layers to apply in the block
            bn_size - Bottleneck size to use in the dense layers
            growth_rate - Growth rate to use in the dense layers
            act_fn - Activation function to use in the dense layers
        """
        super().__init__()
        layers = []
        for layer_idx in range(num_layers):
            layers.append(
                DenseLayer(c_in=c_in + layer_idx * growth_rate, # Input channels are original plus the feature maps from previous layers
                           bn_size=bn_size,
                           growth_rate=growth_rate,
                           act_fn=act_fn)
        )
```

(continues on next page)

(continued from previous page)

```

    self.block = nn.Sequential(*layers)

    def forward(self, x):
        out = self.block(x)
        return out

```

Finally, the TransitionLayer takes as input the final output of a dense block and reduces its channel dimensionality using a 1x1 convolution. To reduce the height and width dimension, we take a slightly different approach than in ResNet and apply an average pooling with kernel size 2 and stride 2. This is because we don't have an additional connection to the output that would consider the full 2x2 patch instead of a single value. Besides, it is more parameter efficient than using a 3x3 convolution with stride 2. Thus, the layer is implemented as follows:

```
[32]: class TransitionLayer(nn.Module):

    def __init__(self, c_in, c_out, act_fn):
        super().__init__()
        self.transition = nn.Sequential(
            nn.BatchNorm2d(c_in),
            act_fn(),
            nn.Conv2d(c_in, c_out, kernel_size=1, bias=False),
            nn.AvgPool2d(kernel_size=2, stride=2) # Average the output for each 2x2_
            ↵pixel group
        )

    def forward(self, x):
        return self.transition(x)
```

Now we can put everything together and create our DenseNet. To specify the number of layers, we use a similar notation as in ResNets and pass on a list of ints representing the number of layers per block. After each dense block except the last one, we apply a transition layer to reduce the dimensionality by 2.

```
[33]: class DenseNet(nn.Module):

    def __init__(self, num_classes=10, num_layers=[6,6,6,6], bn_size=2, growth_rate=16,_
     ↵act_fn_name="relu", **kwargs):
        super().__init__()
        self.hparams = SimpleNamespace(num_classes=num_classes,
                                       num_layers=num_layers,
                                       bn_size=bn_size,
                                       growth_rate=growth_rate,
                                       act_fn_name=act_fn_name,
                                       act_fn=act_fn_by_name[act_fn_name])

        self._create_network()
        self._init_params()

    def _create_network(self):
        c_hidden = self.hparams.growth_rate * self.hparams.bn_size # The start number of_
        ↵hidden channels

        # A first convolution on the original image to scale up the channel size
        self.input_net = nn.Sequential(
            nn.Conv2d(3, c_hidden, kernel_size=3, padding=1) # No batch norm or_
            ↵activation function as done inside the Dense layers
```

(continues on next page)

(continued from previous page)

```

)
# Creating the dense blocks, eventually including transition layers
blocks = []
for block_idx, num_layers in enumerate(self.hparams.num_layers):
    blocks.append(
        DenseBlock(c_in=c_hidden,
                   num_layers=num_layers,
                   bn_size=self.hparams.bn_size,
                   growth_rate=self.hparams.growth_rate,
                   act_fn=self.hparams.act_fn)
    )
    c_hidden = c_hidden + num_layers * self.hparams.growth_rate # Overall output
    ↪ of the dense block
    if block_idx < len(self.hparams.num_layers)-1: # Don't apply transition layer
    ↪ on last block
        blocks.append(
            TransitionLayer(c_in=c_hidden,
                            c_out=c_hidden // 2,
                            act_fn=self.hparams.act_fn))
    c_hidden = c_hidden // 2

self.blocks = nn.Sequential(*blocks)

# Mapping to classification output
self.output_net = nn.Sequential(
    nn.BatchNorm2d(c_hidden), # The features have not passed a non-linearity
    ↪ until here.
    self.hparams.act_fn(),
    nn.AdaptiveAvgPool2d((1,1)),
    nn.Flatten(),
    nn.Linear(c_hidden, self.hparams.num_classes)
)

def __init__(self):
    # Based on our discussion in Tutorial 4, we should initialize the convolutions
    ↪ according to the activation function
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight, nonlinearity=self.hparams.act_fn_name)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)

def forward(self, x):
    x = self.input_net(x)
    x = self.blocks(x)
    x = self.output_net(x)
    return x

```

Let's also add the DenseNet to our model dictionary:

```
[34]: model_dict["DenseNet"] = DenseNet
```

Lastly, we train our network. In contrast to ResNet, DenseNet does not show any issues with Adam, and hence we train it with this optimizer. The other hyperparameters are chosen to result in a network with a similar parameter size as the ResNet and GoogleNet. Commonly, when designing very deep networks, DenseNet is more parameter efficient than ResNet while achieving a similar or even better performance.

```
[35]: densenet_model, densenet_results = train_model(model_name="DenseNet",
                                                 model_hparams={"num_classes": 10,
                                                               "num_layers": [6, 6, 6, 6],
                                                               "bn_size": 2,
                                                               "growth_rate": 16,
                                                               "act_fn_name": "relu"},
                                                 optimizer_name="Adam",
                                                 optimizer_hparams={"lr": 1e-3,
                                                               "weight_decay": 1e-4})
```

GPU available: True, used: True
I1109 15:33:10.534986 139927971620672 distributed.py:49] GPU available: True, used: True
TPU available: False, using: 0 TPU cores
I1109 15:33:10.536422 139927971620672 distributed.py:49] TPU available: False, using: 0
→ TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
I1109 15:33:10.538130 139927971620672 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_
→ VISIBLE_DEVICES: [0]

Found pretrained model at ../saved_models/tutorial5/DenseNet.ckpt, loading...

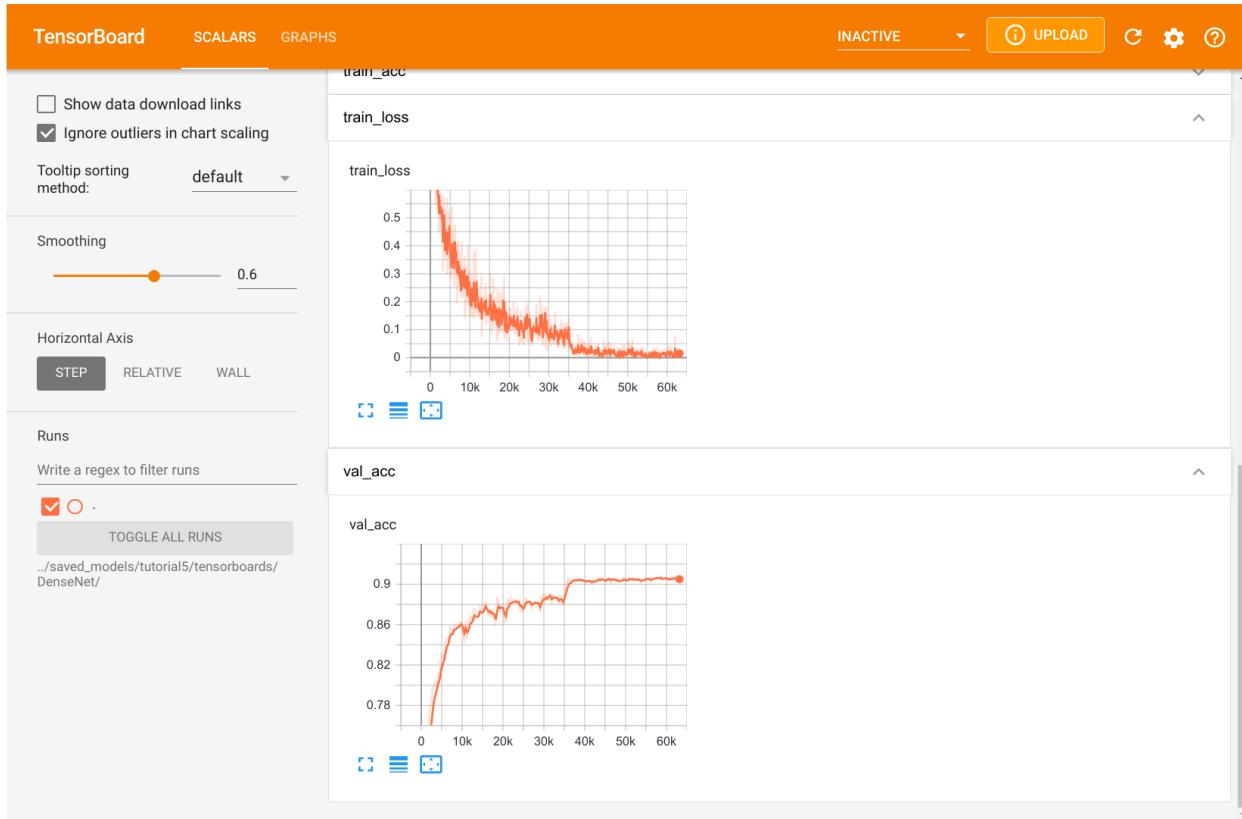
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',
→ layout=Layout(flex='2')), max=...)

HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',
→ layout=Layout(flex='2')), max=...)

Tensorboard log

Finally, we also have another TensorBoard for the DenseNet training. We take a look at it below:

```
[36]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH!
%tensorboard --logdir ../saved_models/tutorial5/tensorboards/DenseNet/
```



The overall course of the validation accuracy and training loss resemble the training of GoogleNet, which is also related to training the network with Adam. Feel free to explore the training metrics yourself.

4.8.5 Conclusion and Comparison

After discussing each model separately, and training all of them, we can finally compare them. First, let's organize the results of all models in a table:

```
[37]: %%html
<!-- Some HTML code to increase font size in the following table -->
<style>
th {font-size: 120%;}
td {font-size: 120%;}
</style>
<IPython.core.display.HTML object>
```

```
[38]: import tabulate
from IPython.display import display, HTML
all_models = [
    ("GoogleNet", googlenet_results, googlenet_model),
    ("ResNet", resnet_results, resnet_model),
    ("ResNetPreAct", resnetpreact_results, resnetpreact_model),
    ("DenseNet", densenet_results, densenet_model)
]
table = [[model_name,
```

(continues on next page)

(continued from previous page)

```

f'{100.0*model_results['val']:4.2f}%',  

f'{100.0*model_results['test']:4.2f}%',  

"{}: {}".format(sum([np.prod(p.shape) for p in model.parameters()]))]  

for model_name, model_results, model in all_models]  

display(HTML(tabulate.tabulate(table, tablefmt='html', headers=["Model", "Val Accuracy",  

    "Test Accuracy", "Num Parameters"])))  

<IPython.core.display.HTML object>

```

First of all, we see that all models are performing reasonably well. Simple models as you have implemented them in the practical achieve considerably lower performance, which is beside the lower number of parameters also attributed to the architecture design choice. GoogleNet is the model to obtain the lowest performance on the validation and test set, although it is very close to DenseNet. A proper hyperparameter search over all the channel sizes in GoogleNet would likely improve the accuracy of the model to a similar level, but this is also expensive given a large number of hyperparameters. ResNet outperforms both DenseNet and GoogleNet by more than 1% on the validation set, while there is a minor difference between both versions, original and pre-activation. We can conclude that for shallow networks, the place of the activation function does not seem to be crucial, although papers have reported the contrary for very deep networks (e.g. He et al.).

In general, we can conclude that ResNet is a simple, but powerful architecture. If we would apply the models on more complex tasks with larger images and more layers inside the networks, we would likely see a bigger gap between GoogleNet and skip-connection architectures like ResNet and DenseNet. A comparison with deeper models on CIFAR10 can be for example found [here](#). Interestingly, DenseNet outperforms the original ResNet on their setup but comes closely behind the Pre-Activation ResNet. The best model, a Dual Path Network (Chen et. al), is actually a combination of ResNet and DenseNet showing that both offer different advantages.

Which model should I choose for my task?

We have reviewed four different models. So, which one should we choose if have given a new task? Usually, starting with a ResNet is a good idea given the superior performance of the CIFAR dataset and its simple implementation. Besides, for the parameter number we have chosen here, ResNet is the fastest as DenseNet and GoogleNet have many more layers that are applied in sequence in our primitive implementation. However, if you have a really difficult task, such as semantic segmentation on HD images, more complex variants of ResNet and DenseNet are recommended.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.9 Tutorial 6: Transformers and Multi-Head Attention

Filled notebook:

Pre-trained models:

Recordings:

JAX+Flax version:**Author:** Phillip Lippe

Note: Interested in JAX? Check out our [JAX+Flax version](#) of this tutorial!

In this tutorial, we will discuss one of the most impactful architectures of the last 2 years: the Transformer model. Since the paper [Attention Is All You Need](#) by Vaswani et al. had been published in 2017, the Transformer architecture has continued to beat benchmarks in many domains, most importantly in Natural Language Processing. Transformers with an incredible amount of parameters can generate long, convincing [essays](#), and opened up new application fields of AI. As the hype of the Transformer architecture seems not to come to an end in the next years, it is important to understand how it works, and have implemented it yourself, which we will do in this notebook.

Despite the huge success of Transformers in NLP, we will *not* include the NLP domain in our notebook here. Why? Firstly, the Master AI at UvA offers many great NLP courses that will take a closer look at the application of the Transformer architecture in NLP ([NLP2, Advanced Topics in Computational Semantics](#)). Secondly, assignment 2 takes already a closer look at language generation on character level, on which you could easily apply our transformer architecture. Finally, and most importantly, there is so much more to the Transformer architecture. NLP is the domain the Transformer architecture has been originally proposed for and had the greatest impact on, but it also accelerated research in other domains, recently even [Computer Vision](#). Thus, we focus here on what makes the Transformer and self-attention so powerful in general. In [Tutorial 15](#), we will discuss the application of Transformers in Computer Vision.

Below, we import our standard libraries. Similarly as in Tutorial 5, we will use [PyTorch Lightning](#) as an additional framework. If you are not familiar with PyTorch Lightning, please make sure to have read Tutorial 5 carefully.

```
[1]: ## Standard libraries
import os
import numpy as np
import random
import math
import json
from functools import partial

## Imports for plotting
import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## tqdm for loading bars
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
```

(continues on next page)

(continued from previous page)

```

import torch.utils.data as data
import torch.optim as optim

## Torchvision
import torchvision
from torchvision.datasets import CIFAR100
from torchvision import transforms

# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial6"

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", device)
Device: cuda:0

```

Two pre-trained models are downloaded below. Make sure to have adjusted your CHECKPOINT_PATH before running this code if not already done.

```

[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial6/"
# Files to download
pretrained_files = ["ReverseTask.ckpt", "SetAnomalyTask.ckpt"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):

```

(continues on next page)

(continued from previous page)

```

file_url = base_url + file_name
print(f"Downloading {file_url}...")
try:
    urllib.request.urlretrieve(file_url, file_path)
except HTTPError as e:
    print("Something went wrong. Please try to download the file from the GDrive_"
    "folder, or contact the author with the full output including the following error:\n",
    e)

```

4.9.1 The Transformer architecture

In the first part of this notebook, we will implement the Transformer architecture by hand. As the architecture is so popular, there already exists a Pytorch module `nn.Transformer` ([documentation](#)) and a [tutorial](#) on how to use it for next token prediction. However, we will implement it here ourselves, to get through to the smallest details.

There are of course many more tutorials out there about attention and Transformers. Below, we list a few that are worth exploring if you are interested in the topic and might want yet another perspective on the topic after this one:

- [Transformer: A Novel Neural Network Architecture for Language Understanding \(Jakob Uszkoreit, 2017\)](#) - The original Google blog post about the Transformer paper, focusing on the application in machine translation.
- [The Illustrated Transformer \(Jay Alammar, 2018\)](#) - A very popular and great blog post intuitively explaining the Transformer architecture with many nice visualizations. The focus is on NLP.
- [Attention? Attention! \(Lilian Weng, 2018\)](#) - A nice blog post summarizing attention mechanisms in many domains including vision.
- [Illustrated: Self-Attention \(Raimi Karim, 2019\)](#) - A nice visualization of the steps of self-attention. Recommended going through if the explanation below is too abstract for you.
- [The Transformer family \(Lilian Weng, 2020\)](#) - A very detailed blog post reviewing more variants of Transformers besides the original one.

What is Attention?

The attention mechanism describes a recent new group of layers in neural networks that has attracted a lot of interest in the past few years, especially in sequence tasks. There are a lot of different possible definitions of “attention” in the literature, but the one we will use here is the following: *the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements’ keys*. So what does this exactly mean? The goal is to take an average over the features of multiple elements. However, instead of weighting each element equally, we want to weight them depending on their actual values. In other words, we want to dynamically decide on which inputs we want to “attend” more than others. In particular, an attention mechanism has usually four parts we need to specify:

- **Query:** The query is a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.
- **Keys:** For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is “offering”, or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.
- **Values:** For each input element, we also have a value vector. This feature vector is the one we want to average over.

- **Score function:** To rate which elements we want to pay attention to, we need to specify a score function f_{attn} . The score function takes the query and a key as input, and output the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product, or a small MLP.

The weights of the average are calculated by a softmax over all score function outputs. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query. If we try to describe it with pseudo-math, we can write:

$$\alpha_i = \frac{\exp(f_{attn}(\text{key}_i, \text{query}))}{\sum_j \exp(f_{attn}(\text{key}_j, \text{query}))}, \quad \text{out} = \sum_i \alpha_i \cdot \text{value}_i$$

Visually, we can show the attention over a sequence of words as follows:

For every word, we have one key and one value vector. The query is compared to all keys with a score function (in this case the dot product) to determine the weights. The softmax is not visualized for simplicity. Finally, the value vectors of all words are averaged using the attention weights.

Most attention mechanisms differ in terms of what queries they use, how the key and value vectors are defined, and what score function is used. The attention applied inside the Transformer architecture is called **self-attention**. In self-attention, each sequence element provides a key, value, and query. For each element, we perform an attention layer where based on its query, we check the similarity of the all sequence elements' keys, and returned a different, averaged value vector for each element. We will now go into a bit more detail by first looking at the specific implementation of the attention mechanism which is in the Transformer case the scaled dot product attention.

Scaled Dot Product Attention

The core concept behind self-attention is the scaled dot product attention. Our goal is to have an attention mechanism with which any element in a sequence can attend to any other while still being efficient to compute. The dot product attention takes as input a set of queries $Q \in \mathbb{R}^{T \times d_k}$, keys $K \in \mathbb{R}^{T \times d_k}$ and values $V \in \mathbb{R}^{T \times d_v}$ where T is the sequence length, and d_k and d_v are the hidden dimensionality for queries/keys and values respectively. For simplicity, we neglect the batch dimension for now. The attention value from element i to j is based on its similarity of the query Q_i and key K_j , using the dot product as the similarity metric. In math, we calculate the dot product attention as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The matrix multiplication QK^T performs the dot product for every possible pair of queries and keys, resulting in a matrix of the shape $T \times T$. Each row represents the attention logits for a specific element i to all other elements in the sequence. On these, we apply a softmax and multiply with the value vector to obtain a weighted mean (the weights being determined by the attention). Another perspective on this attention mechanism offers the computation graph which is visualized below (figure credit - [Vaswani et al., 2017](#)).

One aspect we haven't discussed yet is the scaling factor of $1/\sqrt{d_k}$. This scaling factor is crucial to maintain an appropriate variance of attention values after initialization. Remember that we initialize our layers with the intention of having equal variance throughout the model, and hence, Q and K might also have a variance close to 1. However, performing a dot product over two vectors with a variance σ^2 results in a scalar having d_k -times higher variance:

$$q_i \sim \mathcal{N}(0, \sigma^2), k_i \sim \mathcal{N}(0, \sigma^2) \rightarrow \text{Var}\left(\sum_{i=1}^{d_k} q_i \cdot k_i\right) = \sigma^4 \cdot d_k$$

If we do not scale down the variance back to $\sim \sigma^2$, the softmax over the logits will already saturate to 1 for one random element and 0 for all others. The gradients through the softmax will be close to zero so that we can't learn the parameters

appropriately. Note that the extra factor of σ^2 , i.e., having σ^4 instead of σ^2 , is usually not an issue, since we keep the original variance σ^2 close to 1 anyways.

The block Mask (opt.) in the diagram above represents the optional masking of specific entries in the attention matrix. This is for instance used if we stack multiple sequences with different lengths into a batch. To still benefit from parallelization in PyTorch, we pad the sentences to the same length and mask out the padding tokens during the calculation of the attention values. This is usually done by setting the respective attention logits to a very low value.

After we have discussed the details of the scaled dot product attention block, we can write a function below which computes the output features given the triple of queries, keys, and values:

```
[3]: def scaled_dot_product(q, k, v, mask=None):
    d_k = q.size()[-1]
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits = attn_logits / math.sqrt(d_k)
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -9e15)
    attention = F.softmax(attn_logits, dim=-1)
    values = torch.matmul(attention, v)
    return values, attention
```

Note that our code above supports any additional dimensionality in front of the sequence length so that we can also use it for batches. However, for a better understanding, let's generate a few random queries, keys, and value vectors, and calculate the attention outputs:

```
[4]: seq_len, d_k = 3, 2
pl.seed_everything(42)
q = torch.randn(seq_len, d_k)
k = torch.randn(seq_len, d_k)
v = torch.randn(seq_len, d_k)
values, attention = scaled_dot_product(q, k, v)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Values\n", values)
print("Attention\n", attention)

Q
tensor([[ 0.3367,  0.1288],
       [ 0.2345,  0.2303],
       [-1.1229, -0.1863]])

K
tensor([[ 2.2082, -0.6380],
       [ 0.4617,  0.2674],
       [ 0.5349,  0.8094]])

V
tensor([[ 1.1103, -1.6898],
       [-0.9890,  0.9580],
       [ 1.3221,  0.8172]])

Values
tensor([[ 0.5698, -0.1520],
       [ 0.5379, -0.0265],
       [ 0.2246,  0.5556]])

Attention
tensor([[ 0.4028,  0.2886,  0.3086],
```

(continues on next page)

(continued from previous page)

```
[0.3538, 0.3069, 0.3393],
[0.1303, 0.4630, 0.4067]])
```

Before continuing, make sure you can follow the calculation of the specific values here, and also check it by hand. It is important to fully understand how the scaled dot product attention is calculated.

Multi-Head Attention

The scaled dot product attention allows a network to attend over a sequence. However, often there are multiple different aspects a sequence element wants to attend to, and a single weighted average is not a good option for it. This is why we extend the attention mechanisms to multiple heads, i.e. multiple different query-key-value triplets on the same features. Specifically, given a query, key, and value matrix, we transform those into h sub-queries, sub-keys, and sub-values, which we pass through the scaled dot product attention independently. Afterward, we concatenate the heads and combine them with a final weight matrix. Mathematically, we can express this operation as:

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

We refer to this as Multi-Head Attention layer with the learnable parameters $W_{1\dots h}^Q \in \mathbb{R}^{D \times d_k}$, $W_{1\dots h}^K \in \mathbb{R}^{D \times d_k}$, $W_{1\dots h}^V \in \mathbb{R}^{D \times d_v}$, and $W^O \in \mathbb{R}^{h \cdot d_k \times d_{out}}$ (D being the input dimensionality). Expressed in a computational graph, we can visualize it as below (figure credit - Vaswani et al., 2017).

How are we applying a Multi-Head Attention layer in a neural network, where we don't have an arbitrary query, key, and value vector as input? Looking at the computation graph above, a simple but effective implementation is to set the current feature map in a NN, $X \in \mathbb{R}^{B \times T \times d_{\text{model}}}$, as Q , K and V (B being the batch size, T the sequence length, d_{model} the hidden dimensionality of X). The consecutive weight matrices W^Q , W^K , and W^V can transform X to the corresponding feature vectors that represent the queries, keys, and values of the input. Using this approach, we can implement the Multi-Head Attention module below.

```
[5]: class MultiheadAttention(nn.Module):

    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()
        assert embed_dim % num_heads == 0, "Embedding dimension must be 0 modulo number of heads."

        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        # Stack all weight matrices 1...h together for efficiency
        # Note that in many implementations you see "bias=False" which is optional
        self.qkv_proj = nn.Linear(input_dim, 3*embed_dim)
        self.o_proj = nn.Linear(embed_dim, embed_dim)

        self._reset_parameters()

    def _reset_parameters(self):
        # Original Transformer initialization, see PyTorch documentation
        nn.init.xavier_uniform_(self.qkv_proj.weight)
```

(continues on next page)

(continued from previous page)

```

self.qkv_proj.bias.data.fill_(0)
nn.init.xavier_uniform_(self.o_proj.weight)
self.o_proj.bias.data.fill_(0)

def forward(self, x, mask=None, return_attention=False):
    batch_size, seq_length, _ = x.size()
    qkv = self.qkv_proj(x)

    # Separate Q, K, V from linear output
    qkv = qkv.reshape(batch_size, seq_length, self.num_heads, 3*self.head_dim)
    qkv = qkv.permute(0, 2, 1, 3) # [Batch, Head, SeqLen, Dims]
    q, k, v = qkv.chunk(3, dim=-1)

    # Determine value outputs
    values, attention = scaled_dot_product(q, k, v, mask=mask)
    values = values.permute(0, 2, 1, 3) # [Batch, SeqLen, Head, Dims]
    values = values.reshape(batch_size, seq_length, self.embed_dim)
    o = self.o_proj(values)

    if return_attention:
        return o, attention
    else:
        return o

```

One crucial characteristic of the multi-head attention is that it is permutation-equivariant with respect to its inputs. This means that if we switch two input elements in the sequence, e.g. $X_1 \leftrightarrow X_2$ (neglecting the batch dimension for now), the output is exactly the same besides the elements 1 and 2 switched. Hence, the multi-head attention is actually looking at the input not as a sequence, but as a set of elements. This property makes the multi-head attention block and the Transformer architecture so powerful and widely applicable! But what if the order of the input is actually important for solving the task, like language modeling? The answer is to encode the position in the input features, which we will take a closer look at later (topic *Positional encodings* below).

Before moving on to creating the Transformer architecture, we can compare the self-attention operation with our other common layer competitors for sequence data: convolutions and recurrent neural networks. Below you can find a table by Vaswani et al. (2017) on the complexity per layer, the number of sequential operations, and maximum path length. The complexity is measured by the upper bound of the number of operations to perform, while the maximum path length represents the maximum number of steps a forward or backward signal has to traverse to reach any other position. The lower this length, the better gradient signals can backpropagate for long-range dependencies. Let's take a look at the table below:

n is the sequence length, d is the representation dimension and k is the kernel size of convolutions. In contrast to recurrent networks, the self-attention layer can parallelize all its operations making it much faster to execute for smaller sequence lengths. However, when the sequence length exceeds the hidden dimensionality, self-attention becomes more expensive than RNNs. One way of reducing the computational cost for long sequences is by restricting the self-attention to a neighborhood of inputs to attend over, denoted by r . Nevertheless, there has been recently a lot of work on more efficient Transformer architectures that still allow long dependencies, of which you can find an overview in the paper by Tay et al. (2020) if interested.

Transformer Encoder

Next, we will look at how to apply the multi-head attention block inside the Transformer architecture. Originally, the Transformer model was designed for machine translation. Hence, it got an encoder-decoder structure where the encoder takes as input the sentence in the original language and generates an attention-based representation. On the other hand, the decoder attends over the encoded information and generates the translated sentence in an autoregressive manner, as in a standard RNN. While this structure is extremely useful for Sequence-to-Sequence tasks with the necessity of autoregressive decoding, we will focus here on the encoder part. Many advances in NLP have been made using pure encoder-based Transformer models (if interested, models include the [BERT](#)-family, the [Vision Transformer](#), and more), and in our tutorial, we will also mainly focus on the encoder part. If you have understood the encoder architecture, the decoder is a very small step to implement as well. The full Transformer architecture looks as follows (figure credit - [Vaswani et al., 2017](#)):

The encoder consists of N identical blocks that are applied in sequence. Taking as input x , it is first passed through a Multi-Head Attention block as we have implemented above. The output is added to the original input using a residual connection, and we apply a consecutive Layer Normalization on the sum. Overall, it calculates $\text{LayerNorm}(x + \text{Multihead}(x, x, x))$ (x being Q , K and V input to the attention layer). The residual connection is crucial in the Transformer architecture for two reasons:

1. Similar to ResNets, Transformers are designed to be very deep. Some models contain more than 24 blocks in the encoder. Hence, the residual connections are crucial for enabling a smooth gradient flow through the model.
2. Without the residual connection, the information about the original sequence is lost. Remember that the Multi-Head Attention layer ignores the position of elements in a sequence, and can only learn it based on the input features. Removing the residual connections would mean that this information is lost after the first attention layer (after initialization), and with a randomly initialized query and key vector, the output vectors for position i has no relation to its original input. All outputs of the attention are likely to represent similar/same information, and there is no chance for the model to distinguish which information came from which input element. An alternative option to residual connection would be to fix at least one head to focus on its original input, but this is very inefficient and does not have the benefit of the improved gradient flow.

The Layer Normalization also plays an important role in the Transformer architecture as it enables faster training and provides small regularization. Additionally, it ensures that the features are in a similar magnitude among the elements in the sequence. We are not using Batch Normalization because it depends on the batch size which is often small with Transformers (they require a lot of GPU memory), and BatchNorm has shown to perform particularly bad in language as the features of words tend to have a much higher variance (there are many, very rare words which need to be considered for a good distribution estimate).

Additionally to the Multi-Head Attention, a small fully connected feed-forward network is added to the model, which is applied to each position separately and identically. Specifically, the model uses a Linear→ReLU→Linear MLP. The full transformation including the residual connection can be expressed as:

$$\begin{aligned} \text{FFN}(x) &= \max(0, xW_1 + b_1)W_2 + b_2 \\ x &= \text{LayerNorm}(x + \text{FFN}(x)) \end{aligned}$$

This MLP adds extra complexity to the model and allows transformations on each sequence element separately. You can imagine as this allows the model to “post-process” the new information added by the previous Multi-Head Attention, and prepare it for the next attention block. Usually, the inner dimensionality of the MLP is $2\text{-}8\times$ larger than d_{model} , i.e. the dimensionality of the original input x . The general advantage of a wider layer instead of a narrow, multi-layer MLP is the faster, parallelizable execution.

Finally, after looking at all parts of the encoder architecture, we can start implementing it below. We first start by implementing a single encoder block. Additionally to the layers described above, we will add dropout layers in the MLP and on the output of the MLP and Multi-Head Attention for regularization.

```
[6]: class EncoderBlock(nn.Module):

    def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
        """
        Inputs:
            input_dim - Dimensionality of the input
            num_heads - Number of heads to use in the attention block
            dim_feedforward - Dimensionality of the hidden layer in the MLP
            dropout - Dropout probability to use in the dropout layers
        """
        super().__init__()

        # Attention layer
        self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)

        # Two-layer MLP
        self.linear_net = nn.Sequential(
            nn.Linear(input_dim, dim_feedforward),
            nn.Dropout(dropout),
            nn.ReLU(inplace=True),
            nn.Linear(dim_feedforward, input_dim)
        )

        # Layers to apply in between the main layers
        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Attention part
        attn_out = self.self_attn(x, mask=mask)
        x = x + self.dropout(attn_out)
        x = self.norm1(x)

        # MLP part
        linear_out = self.linear_net(x)
        x = x + self.dropout(linear_out)
        x = self.norm2(x)

    return x
```

Based on this block, we can implement a module for the full Transformer encoder. Additionally to a forward function that iterates through the sequence of encoder blocks, we also provide a function called `get_attention_maps`. The idea of this function is to return the attention probabilities for all Multi-Head Attention blocks in the encoder. This helps us in understanding, and in a sense, explaining the model. However, the attention probabilities should be interpreted with a grain of salt as it does not necessarily reflect the true interpretation of the model (there is a series of papers about this, including [Attention is not Explanation](#) and [Attention is not not Explanation](#)).

```
[7]: class TransformerEncoder(nn.Module):

    def __init__(self, num_layers, **block_args):
        super().__init__()
        self.layers = nn.ModuleList([EncoderBlock(**block_args) for _ in range(num_layers)])
```

(continues on next page)

(continued from previous page)

```

def forward(self, x, mask=None):
    for l in self.layers:
        x = l(x, mask=mask)
    return x

def get_attention_maps(self, x, mask=None):
    attention_maps = []
    for l in self.layers:
        _, attn_map = l.self_attn(x, mask=mask, return_attention=True)
        attention_maps.append(attn_map)
        x = l(x)
    return attention_maps

```

Positional encoding

We have discussed before that the Multi-Head Attention block is permutation-equivariant, and cannot distinguish whether an input comes before another one in the sequence or not. In tasks like language understanding, however, the position is important for interpreting the input words. The position information can therefore be added via the input features. We could learn a embedding for every possible position, but this would not generalize to a dynamical input sequence length. Hence, the better option is to use feature patterns that the network can identify from the features and potentially generalize to larger sequences. The specific pattern chosen by Vaswani et al. are sine and cosine functions of different frequencies, as follows:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{model}}}\right) & \text{if } i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{model}}}\right) & \text{otherwise} \end{cases}$$

$PE_{(pos,i)}$ represents the position encoding at position pos in the sequence, and hidden dimensionality i . These values, concatenated for all hidden dimensions, are added to the original input features (in the Transformer visualization above, see “Positional encoding”), and constitute the position information. We distinguish between even ($i \bmod 2 = 0$) and uneven ($i \bmod 2 = 1$) hidden dimensionalities where we apply a sine/cosine respectively. The intuition behind this encoding is that you can represent $PE_{(pos+k,:)}$ as a linear function of $PE_{(pos,:)}$, which might allow the model to easily attend to relative positions. The wavelengths in different dimensions range from 2π to $10000 \cdot 2\pi$.

The positional encoding is implemented below. The code is taken from the PyTorch tutorial about Transformers on NLP and adjusted for our purposes.

```
[8]: class PositionalEncoding(nn.Module):

    def __init__(self, d_model, max_len=5000):
        """
        Inputs
            d_model - Hidden dimensionality of the input.
            max_len - Maximum length of a sequence to expect.
        """
        super().__init__()

        # Create matrix of [SeqLen, HiddenDim] representing the positional encoding for
        # max_len inputs
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        pe[:, 0::2] = torch.sin(position * (10000 ** (0 / d_model)))
        pe[:, 1::2] = torch.cos(position * (10000 ** (1 / d_model)))
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:x.size(1), :]
        return x
```

(continues on next page)

(continued from previous page)

```

    div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    pe = pe.unsqueeze(0)

    # register_buffer => Tensor which is not a parameter, but should be part of the modules state.
    # Used for tensors that need to be on the same device as the module.
    # persistent=False tells PyTorch to not add the buffer to the state dict (e.g. when we save the model)
    self.register_buffer('pe', pe, persistent=False)

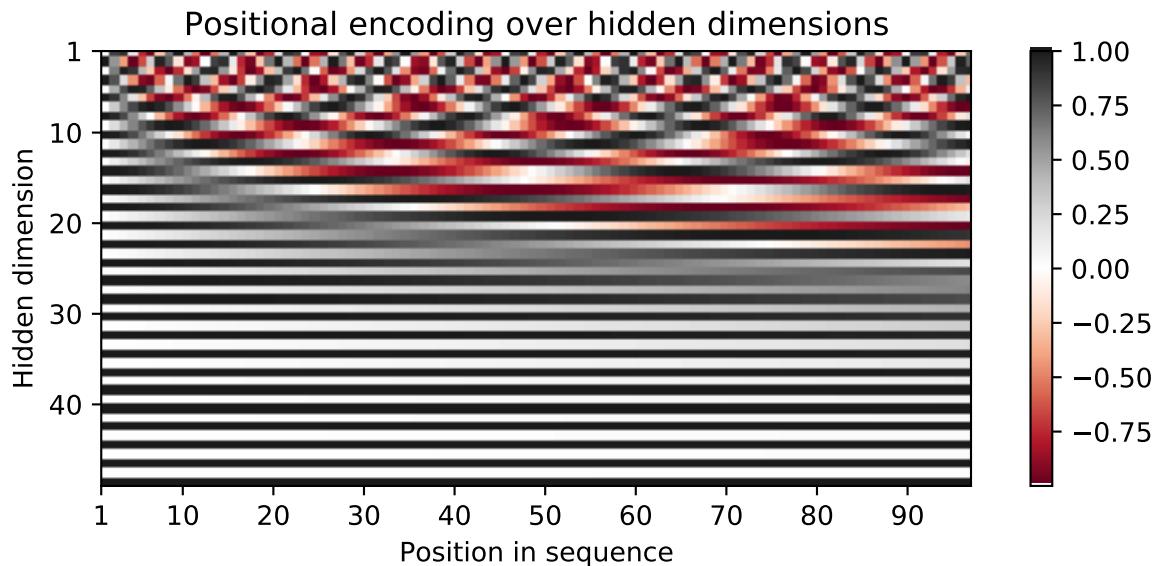
def forward(self, x):
    x = x + self.pe[:, :x.size(1)]
    return x

```

To understand the positional encoding, we can visualize it below. We will generate an image of the positional encoding over hidden dimensionality and position in a sequence. Each pixel, therefore, represents the change of the input feature we perform to encode the specific position. Let's do it below.

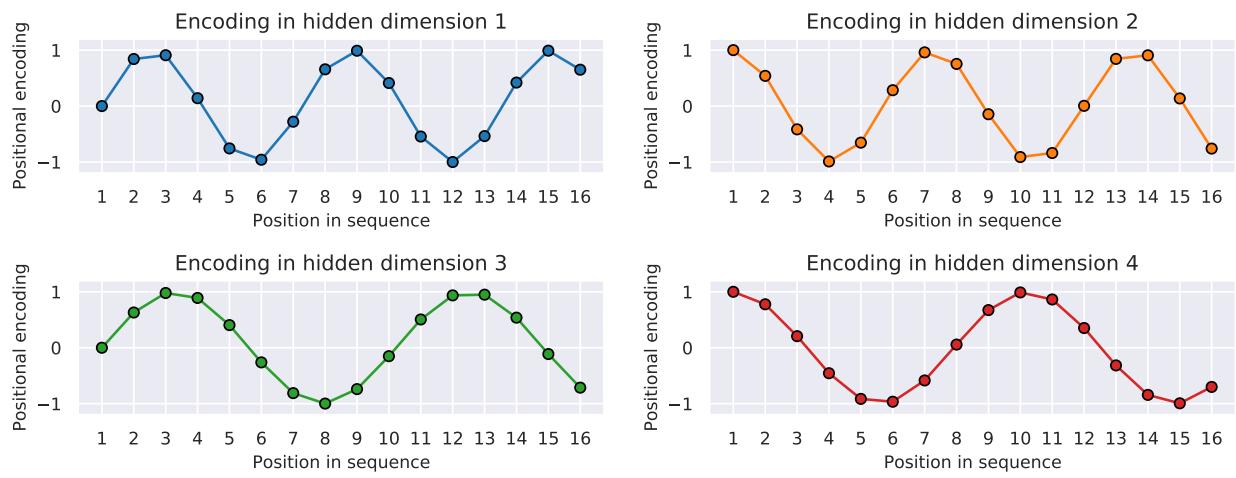
```
[9]: encod_block = PositionalEncoding(d_model=48, max_len=96)
pe = encod_block.pe.squeeze().T.cpu().numpy()

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,3))
pos = ax.imshow(pe, cmap="RdGy", extent=(1,pe.shape[1]+1,pe.shape[0]+1,1))
fig.colorbar(pos, ax=ax)
ax.set_xlabel("Position in sequence")
ax.set_ylabel("Hidden dimension")
ax.set_title("Positional encoding over hidden dimensions")
ax.set_xticks([1]+[i*10 for i in range(1,1+pe.shape[1]//10)])
ax.set_yticks([1]+[i*10 for i in range(1,1+pe.shape[0]//10)])
plt.show()
```



You can clearly see the sine and cosine waves with different wavelengths that encode the position in the hidden dimensions. Specifically, we can look at the sine/cosine wave for each hidden dimension separately, to get a better intuition of the pattern. Below we visualize the positional encoding for the hidden dimensions 1, 2, 3 and 4.

```
[10]: sns.set_theme()
fig, ax = plt.subplots(2, 2, figsize=(12, 4))
ax = [a for a_list in ax for a in a_list]
for i in range(len(ax)):
    ax[i].plot(np.arange(1,17), pe[i,:16], color=f'C{i}', marker="o", markersize=6, markeredgewidth=1)
    ax[i].set_title(f"Encoding in hidden dimension {i+1}")
    ax[i].set_xlabel("Position in sequence", fontsize=10)
    ax[i].set_ylabel("Positional encoding", fontsize=10)
    ax[i].set_xticks(np.arange(1,17))
    ax[i].tick_params(axis='both', which='major', labelsize=10)
    ax[i].tick_params(axis='both', which='minor', labelsize=8)
    ax[i].set_ylim(-1.2, 1.2)
fig.subplots_adjust(hspace=0.8)
sns.reset_orig()
plt.show()
```



As we can see, the patterns between the hidden dimension 1 and 2 only differ in the starting angle. The wavelength is 2π , hence the repetition after position 6. The hidden dimensions 2 and 3 have about twice the wavelength.

Learning rate warm-up

One commonly used technique for training a Transformer is learning rate warm-up. This means that we gradually increase the learning rate from 0 on to our originally specified learning rate in the first few iterations. Thus, we slowly start learning instead of taking very large steps from the beginning. In fact, training a deep Transformer without learning rate warm-up can make the model diverge and achieve a much worse performance on training and testing. Take for instance the following plot by Liu et al. (2019) comparing Adam-vanilla (i.e. Adam without warm-up) vs Adam with a warm-up:

Clearly, the warm-up is a crucial hyperparameter in the Transformer architecture. Why is it so important? There are currently two common explanations. Firstly, Adam uses the bias correction factors which however can lead to a higher variance in the adaptive learning rate during the first iterations. Improved optimizers like RAdam have been shown to overcome this issue, not requiring warm-up for training Transformers. Secondly, the iteratively applied Layer

Normalization across layers can lead to very high gradients during the first iterations, which can be solved by using Pre-Layer Normalization (similar to Pre-Activation ResNet), or replacing Layer Normalization by other techniques (Adaptive Normalization, Power Normalization).

Nevertheless, many applications and papers still use the original Transformer architecture with Adam, because warm-up is a simple, yet effective way of solving the gradient problem in the first iterations. There are many different schedulers we could use. For instance, the original Transformer paper used an exponential decay scheduler with a warm-up. However, the currently most popular scheduler is the cosine warm-up scheduler, which combines warm-up with a cosine-shaped learning rate decay. We can implement it below, and visualize the learning rate factor over epochs.

```
[11]: class CosineWarmupScheduler(optim.lr_scheduler._LRScheduler):
```

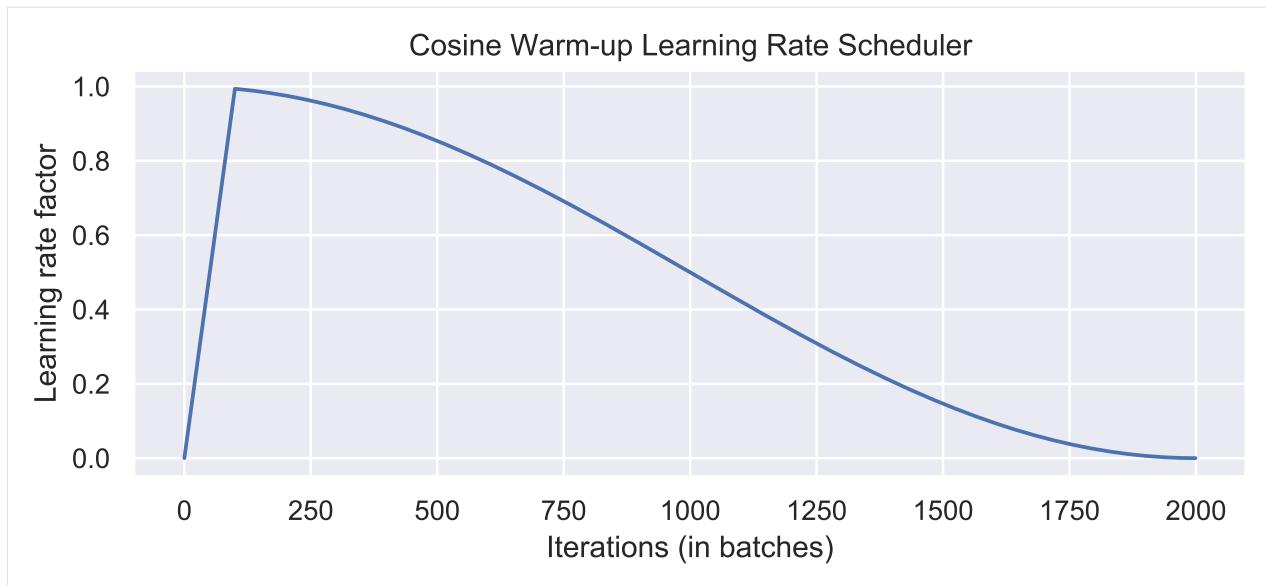
```
    def __init__(self, optimizer, warmup, max_iters):
        self.warmup = warmup
        self.max_num_iters = max_iters
        super().__init__(optimizer)

    def get_lr(self):
        lr_factor = self.get_lr_factor(epoch=self.last_epoch)
        return [base_lr * lr_factor for base_lr in self.base_lrs]

    def get_lr_factor(self, epoch):
        lr_factor = 0.5 * (1 + np.cos(np.pi * epoch / self.max_num_iters))
        if epoch <= self.warmup:
            lr_factor *= epoch * 1.0 / self.warmup
        return lr_factor
```

```
[12]: # Needed for initializing the lr scheduler
p = nn.Parameter(torch.empty(4,4))
optimizer = optim.Adam([p], lr=1e-3)
lr_scheduler = CosineWarmupScheduler(optimizer=optimizer, warmup=100, max_iters=2000)

# Plotting
epochs = list(range(2000))
sns.set()
plt.figure(figsize=(8,3))
plt.plot(epochs, [lr_scheduler.get_lr_factor(e) for e in epochs])
plt.ylabel("Learning rate factor")
plt.xlabel("Iterations (in batches)")
plt.title("Cosine Warm-up Learning Rate Scheduler")
plt.show()
sns.reset_orig()
```



In the first 100 iterations, we increase the learning rate factor from 0 to 1, whereas for all later iterations, we decay it using the cosine wave. Pre-implementations of this scheduler can be found in the popular NLP Transformer library [huggingface](#).

PyTorch Lightning Module

Finally, we can embed the Transformer architecture into a PyTorch lightning module. From Tutorial 5, you know that PyTorch Lightning simplifies our training and test code, as well as structures the code nicely in separate functions. We will implement a template for a classifier based on the Transformer encoder. Thereby, we have a prediction output per sequence element. If we would need a classifier over the whole sequence, the common approach is to add an additional [CLS] token to the sequence, representing the classifier token. However, here we focus on tasks where we have an output per element.

Additionally to the Transformer architecture, we add a small input network (maps input dimensions to model dimensions), the positional encoding, and an output network (transforms output encodings to predictions). We also add the learning rate scheduler, which takes a step each iteration instead of once per epoch. This is needed for the warmup and the smooth cosine decay. The training, validation, and test step is left empty for now and will be filled for our task-specific models.

```
[13]: class TransformerPredictor(pl.LightningModule):
    def __init__(self, input_dim, model_dim, num_classes, num_heads, num_layers, lr,
                 warmup, max_iters, dropout=0.0, input_dropout=0.0):
        """
        Inputs:
            input_dim - Hidden dimensionality of the input
            model_dim - Hidden dimensionality to use inside the Transformer
            num_classes - Number of classes to predict per sequence element
            num_heads - Number of heads to use in the Multi-Head Attention blocks
            num_layers - Number of encoder blocks to use.
            lr - Learning rate in the optimizer
            warmup - Number of warmup steps. Usually between 50 and 500
            max_iters - Number of maximum iterations the model is trained for. This is
                        needed for the CosineWarmup scheduler
        """
        super().__init__()
        self.input_dim = input_dim
        self.model_dim = model_dim
        self.num_classes = num_classes
        self.num_heads = num_heads
        self.num_layers = num_layers
        self.lr = lr
        self.warmup = warmup
        self.max_iters = max_iters
        self.dropout = dropout
        self.input_dropout = input_dropout
        self.cosine_scheduler = CosineWarmupScheduler(max_iters=max_iters, warmup=warmup)
        self.layers = nn.Sequential(*[EncoderLayer(model_dim, num_heads, num_layers) for _ in range(num_layers)])
        self.classifier = nn.Linear(model_dim, num_classes)
```

(continues on next page)

(continued from previous page)

```

dropout - Dropout to apply inside the model
input_dropout - Dropout to apply on the input features
"""

super().__init__()
self.save_hyperparameters()
self._create_model()

def _create_model(self):
    # Input dim -> Model dim
    self.input_net = nn.Sequential(
        nn.Dropout(self.hparams.input_dropout),
        nn.Linear(self.hparams.input_dim, self.hparams.model_dim)
    )
    # Positional encoding for sequences
    self.positional_encoding = PositionalEncoding(d_model=self.hparams.model_dim)
    # Transformer
    self.transformer = TransformerEncoder(num_layers=self.hparams.num_layers,
                                          input_dim=self.hparams.model_dim,
                                          dim_feedforward=2*self.hparams.model_dim,
                                          num_heads=self.hparams.num_heads,
                                          dropout=self.hparams.dropout)
    # Output classifier per sequence element
    self.output_net = nn.Sequential(
        nn.Linear(self.hparams.model_dim, self.hparams.model_dim),
        nn.LayerNorm(self.hparams.model_dim),
        nn.ReLU(inplace=True),
        nn.Dropout(self.hparams.dropout),
        nn.Linear(self.hparams.model_dim, self.hparams.num_classes)
    )

def forward(self, x, mask=None, add_positional_encoding=True):
    """
    Inputs:
        x - Input features of shape [Batch, SeqLen, input_dim]
        mask - Mask to apply on the attention outputs (optional)
        add_positional_encoding - If True, we add the positional encoding to the
        ↪input.
                                         Might not be desired for some tasks.
    """
    x = self.input_net(x)
    if add_positional_encoding:
        x = self.positional_encoding(x)
    x = self.transformer(x, mask=mask)
    x = self.output_net(x)
    return x

@torch.no_grad()
def get_attention_maps(self, x, mask=None, add_positional_encoding=True):
    """
    Function for extracting the attention matrices of the whole Transformer for a
    ↪single batch.
    Input arguments same as the forward pass.
    """

```

(continues on next page)

(continued from previous page)

```

"""
x = self.input_net(x)
if add_positional_encoding:
    x = self.positional_encoding(x)
attention_maps = self.transformer.get_attention_maps(x, mask=mask)
return attention_maps

def configure_optimizers(self):
    optimizer = optim.Adam(self.parameters(), lr=self.hparams.lr)

    # Apply lr scheduler per step
    lr_scheduler = CosineWarmupScheduler(optimizer,
                                         warmup=self.hparams.warmup,
                                         max_iters=self.hparams.max_iters)
    return [optimizer], [{"scheduler": lr_scheduler, "interval": "step"}]

def training_step(self, batch, batch_idx):
    raise NotImplementedError

def validation_step(self, batch, batch_idx):
    raise NotImplementedError

def test_step(self, batch, batch_idx):
    raise NotImplementedError

```

4.9.2 Experiments

After having finished the implementation of the Transformer architecture, we can start experimenting and apply it to various tasks. In this notebook, we will focus on two tasks: parallel Sequence-to-Sequence, and set anomaly detection. The two tasks focus on different properties of the Transformer architecture, and we go through them below.

Sequence to Sequence

A Sequence-to-Sequence task represents a task where the input *and* the output is a sequence, not necessarily of the same length. Popular tasks in this domain include machine translation and summarization. For this, we usually have a Transformer encoder for interpreting the input sequence, and a decoder for generating the output in an autoregressive manner. Here, however, we will go back to a much simpler example task and use only the encoder. Given a sequence of N numbers between 0 and M , the task is to reverse the input sequence. In Numpy notation, if our input is x , the output should be $x[::-1]$. Although this task sounds very simple, RNNs can have issues with such because the task requires long-term dependencies. Transformers are built to support such, and hence, we expect it to perform very well.

First, let's create a dataset class below.

```
[14]: class ReverseDataset(data.Dataset):

    def __init__(self, num_categories, seq_len, size):
        super().__init__()
        self.num_categories = num_categories
        self.seq_len = seq_len
        self.size = size
```

(continues on next page)

(continued from previous page)

```

self.data = torch.randint(self.num_categories, size=(self.size, self.seq_len))

def __len__(self):
    return self.size

def __getitem__(self, idx):
    inp_data = self.data[idx]
    labels = torch.flip(inp_data, dims=(0,))
    return inp_data, labels

```

We create an arbitrary number of random sequences of numbers between 0 and num_categories-1. The label is simply the tensor flipped over the sequence dimension. We can create the corresponding data loaders below.

```
[15]: dataset = partial(ReverseDataset, 10, 16)
train_loader = data.DataLoader(dataset(50000), batch_size=128, shuffle=True, drop_
˓last=True, pin_memory=True)
val_loader = data.DataLoader(dataset(1000), batch_size=128)
test_loader = data.DataLoader(dataset(10000), batch_size=128)
```

Let's look at an arbitrary sample of the dataset:

```
[16]: inp_data, labels = train_loader.dataset[0]
print("Input data:", inp_data)
print("Labels: ", labels)

Input data: tensor([9, 6, 2, 0, 6, 2, 7, 9, 7, 3, 3, 4, 3, 7, 0, 9])
Labels: tensor([9, 0, 7, 3, 4, 3, 3, 7, 9, 7, 2, 6, 0, 2, 6, 9])
```

During training, we pass the input sequence through the Transformer encoder and predict the output for each input token. We use the standard Cross-Entropy loss to perform this. Every number is represented as a one-hot vector. Remember that representing the categories as single scalars decreases the expressiveness of the model extremely as 0 and 1 are not closer related than 0 and 9 in our example. An alternative to a one-hot vector is using a learned embedding vector as it is provided by the PyTorch module `nn.Embedding`. However, using a one-hot vector with an additional linear layer as in our case has the same effect as an embedding layer (`self.input_net` maps one-hot vector to a dense vector, where each row of the weight matrix represents the embedding for a specific category).

To implement the training dynamic, we create a new class inheriting from `TransformerPredictor` and overwriting the training, validation and test step functions.

```
[17]: class ReversePredictor(TransformerPredictor):

    def _calculate_loss(self, batch, mode="train"):
        # Fetch data and transform categories to one-hot vectors
        inp_data, labels = batch
        inp_data = F.one_hot(inp_data, num_classes=self.hparams.num_classes).float()

        # Perform prediction and calculate loss and accuracy
        preds = self.forward(inp_data, add_positional_encoding=True)
        loss = F.cross_entropy(preds.view(-1,preds.size(-1)), labels.view(-1))
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        # Logging
        self.log(f"{mode}_loss", loss)
        self.log(f"{mode}_acc", acc)
```

(continues on next page)

(continued from previous page)

```

    return loss, acc

def training_step(self, batch, batch_idx):
    loss, _ = self._calculate_loss(batch, mode="train")
    return loss

def validation_step(self, batch, batch_idx):
    _ = self._calculate_loss(batch, mode="val")

def test_step(self, batch, batch_idx):
    _ = self._calculate_loss(batch, mode="test")

```

Finally, we can create a training function similar to the one we have seen in Tutorial 5 for PyTorch Lightning. We create a `pl.Trainer` object, running for N epochs, logging in TensorBoard, and saving our best model based on the validation. Afterward, we test our models on the test set. An additional parameter we pass to the trainer here is `gradient_clip_val`. This clips the norm of the gradients for all parameters before taking an optimizer step and prevents the model from diverging if we obtain very high gradients at, for instance, sharp loss surfaces (see many good blog posts on gradient clipping, like [DeepAI glossary](#)). For Transformers, gradient clipping can help to further stabilize the training during the first few iterations, and also afterward. In plain PyTorch, you can apply gradient clipping via `torch.nn.utils.clip_grad_norm_(...)` (see [documentation](#)). The clip value is usually between 0.5 and 10, depending on how harsh you want to clip large gradients. After having explained this, let's implement the training function:

```
[18]: def train_reverse(**kwargs):
    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "ReverseTask")
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                          callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
                           monitor="val_acc")],
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=10,
                          gradient_clip_val=5)
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "ReverseTask.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = ReversePredictor.load_from_checkpoint(pretrained_filename)
    else:
        model = ReversePredictor(max_iters=trainer.max_epochs*len(train_loader), **kwargs)
    trainer.fit(model, train_loader, val_loader)

    # Test best model on validation and test set
    val_result = trainer.test(model, val_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"test_acc": test_result[0]["test_acc"], "val_acc": val_result[0]["test_acc"]}

```

(continues on next page)

(continued from previous page)

```
model = model.to(device)
return model, result
```

Finally, we can train the model. In this setup, we will use a single encoder block and a single head in the Multi-Head Attention. This is chosen because of the simplicity of the task, and in this case, the attention can actually be interpreted as an “explanation” of the predictions (compared to the other papers above dealing with deep Transformers).

```
[19]: reverse_model, reverse_result = train_reverse(input_dim=train_loader.dataset.num_
↪categories,
↪model_dim=32,
↪num_heads=1,
↪num_classes=train_loader.dataset.num_
↪categories,
↪num_layers=1,
↪dropout=0.0,
↪lr=5e-4,
↪warmup=50)
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Found pretrained model, loading...
```

```
/home/phillip/anaconda3/envs/nlp1/lib/python3.7/site-packages/pytorch_lightning/
↪utilities/distributed.py:45: UserWarning: The dataloader, test dataloader 0, does not_
↪have many workers which may be a bottleneck. Consider increasing the value of the `num_
↪workers` argument` (try 16 which is the number of cpus on this machine) in the_
↪`DataLoader` init to improve performance.
warnings.warn(*args, **kwargs)
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',_
↪layout=Layout(flex='2')), max=...))
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',_
↪layout=Layout(flex='2')), max=...))
```

The warning of PyTorch Lightning regarding the number of workers can be ignored for now. As the data set is so simple and the `__getitem__` finishes a neglectable time, we don't need subprocesses to provide us the data (in fact, more workers can slow down the training as we have communication overhead among processes/threads). First, let's print the results:

```
[20]: print(f"Val accuracy: {(100.0 * reverse_result['val_acc']):.2f}%")
print(f"Test accuracy: {(100.0 * reverse_result['test_acc']):.2f}%")
```

```
Val accuracy: 100.00%
```

```
Test accuracy: 100.00%
```

As we would have expected, the Transformer can correctly solve the task. However, how does the attention in the Multi-Head Attention block looks like for an arbitrary input? Let's try to visualize it below.

```
[21]: data_input, labels = next(iter(val_loader))
inp_data = F.one_hot(data_input, num_classes=reverse_model.hparams.num_classes).float()
inp_data = inp_data.to(device)
attention_maps = reverse_model.get_attention_maps(inp_data)
```

The object `attention_maps` is a list of length N where N is the number of layers. Each element is a tensor of shape [Batch, Heads, SeqLen, SeqLen], which we can verify below.

```
[22]: attention_maps[0].shape
[22]: torch.Size([128, 1, 16, 16])
```

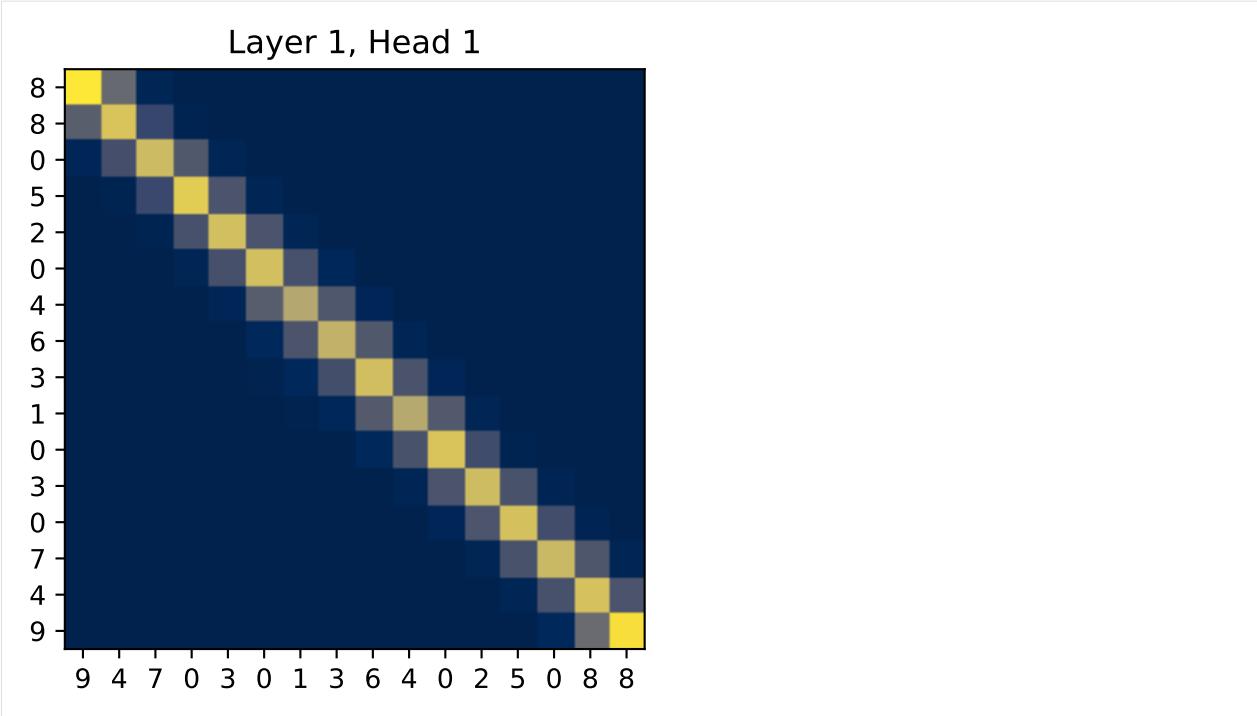
Next, we will write a plotting function that takes as input the sequences, attention maps, and an index indicating for which batch element we want to visualize the attention map. We will create a plot where over rows, we have different layers, while over columns, we show the different heads. Remember that the softmax has been applied for each row separately.

```
[23]: def plot_attention_maps(input_data, attn_maps, idx=0):
    if input_data is not None:
        input_data = input_data[idx].detach().cpu().numpy()
    else:
        input_data = np.arange(attn_maps[0][idx].shape[-1])
    attn_maps = [m[idx].detach().cpu().numpy() for m in attn_maps]

    num_heads = attn_maps[0].shape[0]
    num_layers = len(attn_maps)
    seq_len = input_data.shape[0]
    fig_size = 4 if num_heads == 1 else 3
    fig, ax = plt.subplots(num_layers, num_heads, figsize=(num_heads*fig_size, num_
    ↪layers*fig_size))
    if num_layers == 1:
        ax = [ax]
    if num_heads == 1:
        ax = [[a] for a in ax]
    for row in range(num_layers):
        for column in range(num_heads):
            ax[row][column].imshow(attn_maps[row][column], origin='lower', vmin=0)
            ax[row][column].set_xticks(list(range(seq_len)))
            ax[row][column].set_xticklabels(input_data.tolist())
            ax[row][column].set_yticks(list(range(seq_len)))
            ax[row][column].set_yticklabels(input_data.tolist())
            ax[row][column].set_title(f"Layer {row+1}, Head {column+1}")
    fig.subplots_adjust(hspace=0.5)
    plt.show()
```

Finally, we can plot the attention map of our trained Transformer on the reverse task:

```
[24]: plot_attention_maps(data_input, attention_maps, idx=0)
```

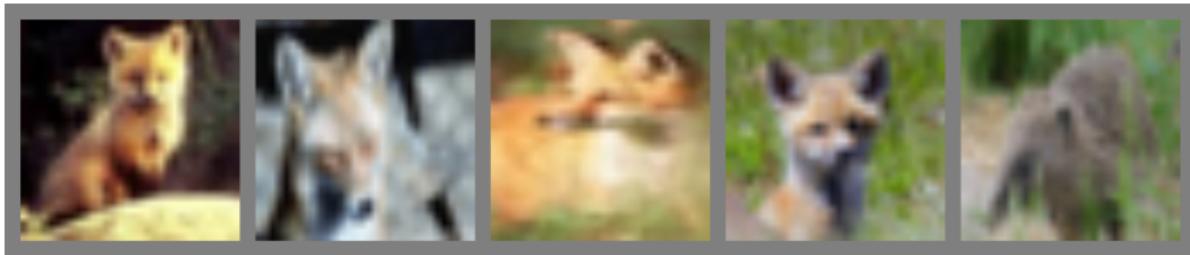


The model has learned to attend to the token that is on the flipped index of itself. Hence, it actually does what we intended it to do. We see that it however also pays some attention to values close to the flipped index. This is because the model doesn't need the perfect, hard attention to solve this problem, but is fine with this approximate, noisy attention map. The close-by indices are caused by the similarity of the positional encoding, which we also intended with the positional encoding.

Set Anomaly Detection

Besides sequences, sets are another data structure that is relevant for many applications. In contrast to sequences, elements are unordered in a set. RNNs can only be applied on sets by assuming an order in the data, which however biases the model towards a non-existing order in the data. [Vinyals et al. \(2015\)](#) and other papers have shown that the assumed order can have a significant impact on the model's performance, and hence, we should try to not use RNNs on sets. Ideally, our model should be permutation-equivariant/invariant such that the output is the same no matter how we sort the elements in a set.

Transformers offer the perfect architecture for this as the Multi-Head Attention is permutation-equivariant, and thus, outputs the same values no matter in what order we enter the inputs (inputs and outputs are permuted equally). The task we are looking at for sets is *Set Anomaly Detection* which means that we try to find the element(s) in a set that does not fit the others. In the research community, the common application of anomaly detection is performed on a set of images, where $N - 1$ images belong to the same category/have the same high-level features while one belongs to another category. Note that category does not necessarily have to relate to a class in a standard classification problem, but could be the combination of multiple features. For instance, on a face dataset, this could be people with glasses, male, beard, etc. An example of distinguishing different animals can be seen below. The first four images show foxes, while the last represents a different animal. We want to recognize that the last image shows a different animal, but it is not relevant which class of animal it is.



In this tutorial, we will use the CIFAR100 dataset. CIFAR100 has 600 images for 100 classes each with a resolution of 32x32, similar to CIFAR10. The larger amount of classes requires the model to attend to specific features in the images instead of coarse features as in CIFAR10, therefore making the task harder. We will show the model a set of 9 images of one class, and 1 image from another class. The task is to find the image that is from a different class than the other images. Using the raw images directly as input to the Transformer is not a good idea, because it is not translation invariant as a CNN, and would need to learn to detect image features from high-dimensional input first of all. Instead, we will use a pre-trained ResNet34 model from the torchvision package to obtain high-level, low-dimensional features of the images. The ResNet model has been pre-trained on the [ImageNet](#) dataset which contains 1 million images of 1k classes and varying resolutions. However, during training and testing, the images are usually scaled to a resolution of 224x224, and hence we rescale our CIFAR images to this resolution as well. Below, we will load the dataset, and prepare the data for being processed by the ResNet model.

```
[25]: # ImageNet statistics
DATA_MEANS = np.array([0.485, 0.456, 0.406])
DATA_STD = np.array([0.229, 0.224, 0.225])
# As torch tensors for later preprocessing
TORCH_DATA_MEANS = torch.from_numpy(DATA_MEANS).view(1,3,1,1)
TORCH_DATA_STD = torch.from_numpy(DATA_STD).view(1,3,1,1)

# Resize to 224x224, and normalize to ImageNet statistic
transform = transforms.Compose([transforms.Resize((224,224)),
                               transforms.ToTensor(),
                               transforms.Normalize(DATA_MEANS, DATA_STD)
                             ])
# Loading the training dataset.
train_set = CIFAR100(root=DATASET_PATH, train=True, transform=transform, download=True)

# Loading the test set
test_set = CIFAR100(root=DATASET_PATH, train=False, transform=transform, download=True)

Files already downloaded and verified
Files already downloaded and verified
```

Next, we want to run the pre-trained ResNet model on the images, and extract the features before the classification layer. These are the most high-level features, and should sufficiently describe the images. CIFAR100 has some similarity to ImageNet, and thus we are not retraining the ResNet model in any form. However, if you would want to get the best performance and have a very large dataset, it would be better to add the ResNet to the computation graph during training and finetune its parameters as well. As we don't have a large enough dataset and want to train our model efficiently, we will extract the features beforehand. Let's load and prepare the model below.

```
[26]: import os
os.environ["TORCH_HOME"] = CHECKPOINT_PATH
pretrained_model = torchvision.models.resnet34(weights='IMAGENET1K_V1')
# Remove classification layer
# In some models, it is called "fc", others have "classifier"
```

(continues on next page)

(continued from previous page)

```
# Setting both to an empty sequential represents an identity map of the final features.
pretrained_model.fc = nn.Sequential()
pretrained_model.classifier = nn.Sequential()
# To GPU
pretrained_model = pretrained_model.to(device)

# Only eval, no gradient required
pretrained_model.eval()
for p in pretrained_model.parameters():
    p.requires_grad = False
```

We will now write a extraction function for the features below. This cell requires access to a GPU, as the model is rather deep and the images relatively large. The GPUs on GoogleColab are sufficient, but running this cell can take 2-3 minutes. Once it is run, the features are exported on disk so they don't have to be recalculated every time you run the notebook. However, this requires >150MB free disk space. So it is recommended to run this only on a local computer if you have enough free disk and a GPU (GoogleColab is fine for this). If you do not have a GPU, you can download the features from the [GoogleDrive folder](#).

```
[27]: @torch.no_grad()
def extract_features(dataset, save_file):
    if not os.path.isfile(save_file):
        data_loader = data.DataLoader(dataset, batch_size=128, shuffle=False, drop_
    ↵last=False, num_workers=4)
        extracted_features = []
        for imgs, _ in tqdm(data_loader):
            imgs = imgs.to(device)
            feats = pretrained_model(imgs)
            extracted_features.append(feats)
        extracted_features = torch.cat(extracted_features, dim=0)
        extracted_features = extracted_features.detach().cpu()
        torch.save(extracted_features, save_file)
    else:
        extracted_features = torch.load(save_file)
    return extracted_features

train_feat_file = os.path.join(CHECKPOINT_PATH, "train_set_features.tar")
train_set_feats = extract_features(train_set, train_feat_file)

test_feat_file = os.path.join(CHECKPOINT_PATH, "test_set_features.tar")
test_feats = extract_features(test_set, test_feat_file)
```

Let's verify the feature shapes below. The training should have 50k elements, and the test 10k images. The feature dimension is 512 for the ResNet34. If you experiment with other models, you likely see a different feature dimension.

```
[28]: print("Train:", train_set_feats.shape)
print("Test: ", test_feats.shape)

Train: torch.Size([50000, 512])
Test:  torch.Size([10000, 512])
```

As usual, we want to create a validation set to detect when we should stop training. In this case, we will split the training set into 90% training, 10% validation. However, the difficulty is here that we need to ensure that the validation set has the same number of images for all 100 labels. Otherwise, we have a class imbalance which is not good for creating the image sets. Hence, we take 10% of the images for each class, and move them into the validation set. The code below

does exactly this.

```
[29]: ## Split train into train+val
# Get labels from train set
labels = train_set.targets

# Get indices of images per class
labels = torch.LongTensor(labels)
num_labels = labels.max() + 1
sorted_indices = torch.argsort(labels).reshape(num_labels, -1) # [classes, num_imgs_per_class]

# Determine number of validation images per class
num_val_exmps = sorted_indices.shape[1] // 10

# Get image indices for validation and training
val_indices = sorted_indices[:, :num_val_exmps].reshape(-1)
train_indices = sorted_indices[:, num_val_exmps:].reshape(-1)

# Group corresponding image features and labels
train_feats, train_labels = train_set.feats[train_indices], labels[train_indices]
val_feats, val_labels = train_set.feats[val_indices], labels[val_indices]
```

Now we can prepare a dataset class for the set anomaly task. We define an epoch to be the sequence in which each image has been exactly once as an “anomaly”. Hence, the length of the dataset is the number of images in it. For the training set, each time we access an item with `__getitem__`, we sample a random, different class than the image at the corresponding index `idx` has. In a second step, we sample $N - 1$ images of this sampled class. The set of 10 images is finally returned. The randomness in the `__getitem__` allows us to see a slightly different set during each iteration. However, we can't use the same strategy for the test set as we want the test dataset to be the same every time we iterate over it. Hence, we sample the sets in the `__init__` method, and return those in `__getitem__`. The code below implements exactly this dynamic.

```
[30]: class SetAnomalyDataset(data.Dataset):

    def __init__(self, img_feats, labels, set_size=10, train=True):
        """
        Inputs:
            img_feats - Tensor of shape [num_imgs, img_dim]. Represents the high-level features.
            labels - Tensor of shape [num_imgs], containing the class labels for the images
            set_size - Number of elements in a set. N-1 are sampled from one class, and one from another one.
            train - If True, a new set will be sampled every time __getitem__ is called.
        """
        super().__init__()
        self.img_feats = img_feats
        self.labels = labels
        self.set_size = set_size - 1 # The set size is here the size of correct images
        self.train = train

        # Tensors with indices of the images per class
        self.num_labels = labels.max() + 1
        self.img_idx_by_label = torch.argsort(self.labels).reshape(self.num_labels, -1)
```

(continues on next page)

(continued from previous page)

```

    if not train:
        self.test_sets = self._create_test_sets()

    def _create_test_sets(self):
        # Pre-generates the sets for each image for the test set
        test_sets = []
        num_imgs = self.img_feats.shape[0]
        np.random.seed(42)
        test_sets = [self.sample_img_set(self.labels[idx]) for idx in range(num_imgs)]
        test_sets = torch.stack(test_sets, dim=0)
        return test_sets

    def sample_img_set(self, anomaly_label):
        """
        Samples a new set of images, given the label of the anomaly.
        The sampled images come from a different class than anomaly_label
        """
        # Sample class from 0,...,num_classes-1 while skipping anomaly_label as class
        set_label = np.random.randint(self.num_labels-1)
        if set_label >= anomaly_label:
            set_label += 1

        # Sample images from the class determined above
        img_indices = np.random.choice(self.img_idx_by_label.shape[1], size=self.set_
        ↵size, replace=False)
        img_indices = self.img_idx_by_label[set_label, img_indices]
        return img_indices

    def __len__(self):
        return self.img_feats.shape[0]

    def __getitem__(self, idx):
        anomaly = self.img_feats[idx]
        if self.train: # If train => sample
            img_indices = self.sample_img_set(self.labels[idx])
        else: # If test => use pre-generated ones
            img_indices = self.test_sets[idx]

        # Concatenate images. The anomaly is always the last image for simplicity
        img_set = torch.cat([self.img_feats[img_indices], anomaly[None]], dim=0)
        indices = torch.cat([img_indices, torch.LongTensor([idx])], dim=0)
        label = img_set.shape[0]-1

        # We return the indices of the images for visualization purpose. "Label" is the
        ↵index of the anomaly
        return img_set, indices, label

```

Next, we can setup our datasets and data loaders below. Here, we will use a set size of 10, i.e. 9 images from one

category + 1 anomaly. Feel free to change it if you want to experiment with the sizes.

```
[31]: SET_SIZE = 10
test_labels = torch.LongTensor(test_set.targets)

train_anom_dataset = SetAnomalyDataset(train_feats, train_labels, set_size=SET_SIZE, ↴
↪ train=True)
val_anom_dataset = SetAnomalyDataset(val_feats, val_labels, set_size=SET_SIZE, ↴
↪ train=False)
test_anom_dataset = SetAnomalyDataset(test_feats, test_labels, set_size=SET_SIZE, ↴
↪ train=False)

train_anom_loader = data.DataLoader(train_anom_dataset, batch_size=64, shuffle=True, ↴
↪ drop_last=True, num_workers=4, pin_memory=True)
val_anom_loader = data.DataLoader(val_anom_dataset, batch_size=64, shuffle=False, ↴
↪ drop_last=False, num_workers=4)
test_anom_loader = data.DataLoader(test_anom_dataset, batch_size=64, shuffle=False, ↴
↪ drop_last=False, num_workers=4)
```

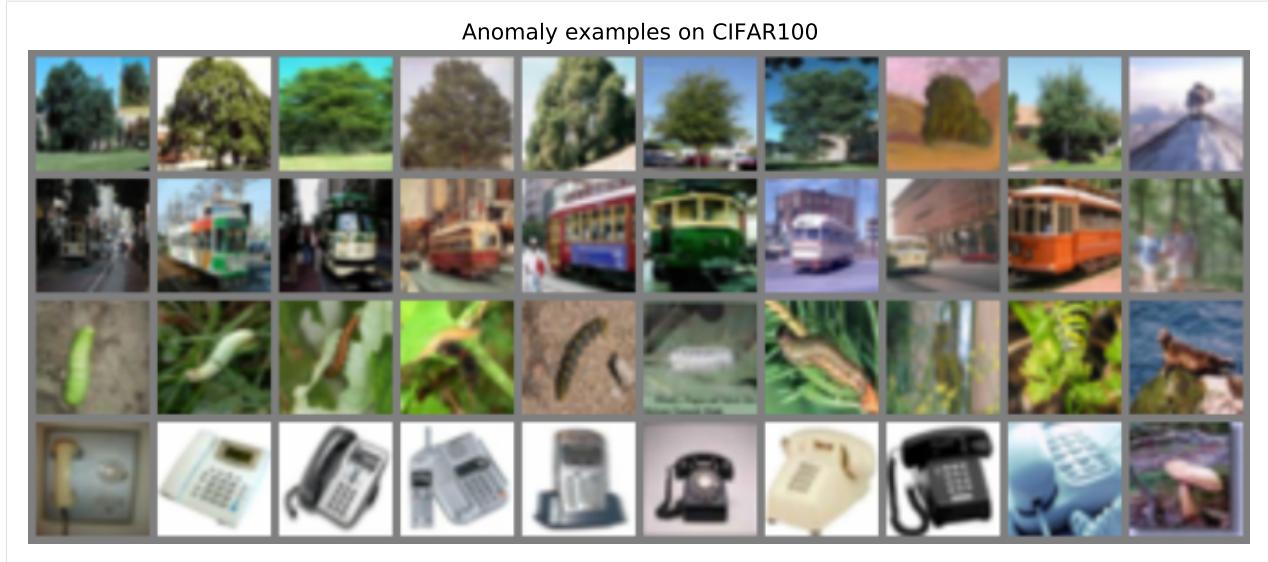
To understand the dataset a little better, we can plot below a few sets from the test dataset. Each row shows a different input set, where the first 9 are from the same class.

```
[32]: def visualize_exmp(indices, orig_dataset):
    images = [orig_dataset[idx] for idx in indices.reshape(-1)]
    images = torch.stack(images, dim=0)
    images = images * TORCH_DATA_STD + TORCH_DATA_MEANS

    img_grid = torchvision.utils.make_grid(images, nrow=SET_SIZE, normalize=True, pad_
↪ value=0.5, padding=16)
    img_grid = img_grid.permute(1, 2, 0)

    plt.figure(figsize=(12,8))
    plt.title("Anomaly examples on CIFAR100")
    plt.imshow(img_grid)
    plt.axis('off')
    plt.show()
    plt.close()

_, indices, _ = next(iter(test_anom_loader))
visualize_exmp(indices[:4], test_set)
```



We can already see that for some sets the task might be easier than for others. Difficulties can especially arise if the anomaly is in a different, but yet visually similar class (e.g. train vs bus, flour vs worm, etc.).

After having prepared the data, we can look closer at the model. Here, we have a classification of the whole set. For the prediction to be permutation-equivariant, we will output one logit for each image. Over these logits, we apply a softmax and train the anomaly image to have the highest score/probability. This is a bit different than a standard classification layer as the softmax is applied over images, not over output classes in the classical sense. However, if we swap two images in their position, we effectively swap their position in the output softmax. Hence, the prediction is equivariant with respect to the input. We implement this idea below in the subclass of the Transformer Lightning module.

```
[33]: class AnomalyPredictor(TransformerPredictor):

    def _calculate_loss(self, batch, mode="train"):
        img_sets, _, labels = batch
        preds = self.forward(img_sets, add_positional_encoding=False) # No positional
        ↪encodings as it is a set, not a sequence!
        preds = preds.squeeze(dim=-1) # Shape: [Batch_size, set_size]
        loss = F.cross_entropy(preds, labels) # Softmax/CE over set dimension
        acc = (preds.argmax(dim=-1) == labels).float().mean()
        self.log(f"{mode}_loss", loss)
        self.log(f"{mode}_acc", acc, on_step=False, on_epoch=True)
        return loss, acc

    def training_step(self, batch, batch_idx):
        loss, _ = self._calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="val")

    def test_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="test")
```

Finally, we write our train function below. It has the exact same structure as the reverse task one, hence not much of an explanation is needed here.

```
[34]: def train_anomaly(**kwargs):
    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "SetAnomalyTask")
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                          callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
                           monitor="val_acc")],
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=100,
                          gradient_clip_val=2)
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "SetAnomalyTask.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = AnomalyPredictor.load_from_checkpoint(pretrained_filename)
    else:
        model = AnomalyPredictor(max_iters=trainer.max_epochs*len(train_anom_loader), **kwargs)
        trainer.fit(model, train_anom_loader, val_anom_loader)
        model = AnomalyPredictor.load_from_checkpoint(trainer.checkpoint_callback.best_model_path)

    # Test best model on validation and test set
    train_result = trainer.test(model, train_anom_loader, verbose=False)
    val_result = trainer.test(model, val_anom_loader, verbose=False)
    test_result = trainer.test(model, test_anom_loader, verbose=False)
    result = {"test_acc": test_result[0]["test_acc"], "val_acc": val_result[0]["test_acc"],
              "train_acc": train_result[0]["test_acc"]}

    model = model.to(device)
    return model, result
```

Let's finally train our model. We will use 4 layers with 4 attention heads each. The hidden dimensionality of the model is 256, and we use a dropout of 0.1 throughout the model for good regularization. Note that we also apply the dropout on the input features, as this makes the model more robust against image noise and generalizes better. Again, we use warmup to slowly start our model training.

```
[35]: anomaly_model, anomaly_result = train_anomaly(input_dim=train_anom_dataset.img_feats.
                                                 .shape[-1],
                                                 model_dim=256,
                                                 num_heads=4,
                                                 num_classes=1,
                                                 num_layers=4,
                                                 dropout=0.1,
                                                 input_dropout=0.1,
                                                 lr=5e-4,
                                                 warmup=100)
```

GPU available: True, used: True

(continues on next page)

(continued from previous page)

```

WARNING: Logging before flag parsing goes to stderr.
I1109 10:43:31.036801 139648634296128 distributed.py:49] GPU available: True, used: True
TPU available: False, using: 0 TPU cores
I1109 10:43:31.038146 139648634296128 distributed.py:49] TPU available: False, using: 0 ↵
    ↵ TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
I1109 10:43:31.039162 139648634296128 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_ ↵
    ↵ VISIBLE_DEVICES: [0]

Found pretrained model, loading...

/home/phillip/anaconda3/envs/nlp1/lib/python3.7/site-packages/pytorch_lightning/ ↵
    ↵ utilities/distributed.py:45: UserWarning: Your test_dataloader has `shuffle=True`, it ↵
    ↵ is best practice to turn this off for validation and test dataloaders. ↵
    ↵ warnings.warn(*args, **kwargs)

HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', ↵
    ↵ layout=Layout(flex='2')), max=...)

HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', ↵
    ↵ layout=Layout(flex='2')), max=...)

HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', ↵
    ↵ layout=Layout(flex='2')), max=...)

```

We can print the achieved accuracy below.

```
[36]: print(f"Train accuracy: {100.0*anomaly_result['train_acc']):4.2f}%")
print(f"Val accuracy: {100.0*anomaly_result['val_acc']):4.2f}%")
print(f"Test accuracy: {100.0*anomaly_result['test_acc']):4.2f}%)"

Train accuracy: 97.77%
Val accuracy: 94.38%
Test accuracy: 94.30%
```

With ~94% validation and test accuracy, the model generalizes quite well. It should be noted that you might see slightly different scores depending on what computer/device you are running this notebook. This is because despite setting the seed before generating the test dataset, it is not the same across platforms and numpy versions. Nevertheless, we can conclude that the model performs quite well and can solve the task for most sets. Before trying to interpret the model, let's verify that our model is permutation-equivariant, and assigns the same predictions for different permutations of the input set. For this, we sample a batch from the test set and run it through the model to obtain the probabilities.

```
[37]: inp_data, indices, labels = next(iter(test_anom_loader))
inp_data = inp_data.to(device)

anomaly_model.eval()

with torch.no_grad():
    preds = anomaly_model.forward(inp_data, add_positional_encoding=False)
    preds = F.softmax(preds.squeeze(dim=-1), dim=-1)
```

(continues on next page)

(continued from previous page)

```
# Permut input data
permut = np.random.permutation(inp_data.shape[1])
perm_inp_data = inp_data[:,permut]
perm_preds = anomaly_model.forward(perm_inp_data, add_positional_encoding=False)
perm_preds = F.softmax(perm_preds.squeeze(dim=-1), dim=-1)

assert (preds[:,permut] - perm_preds).abs().max() < 1e-5, "Predictions are not \
permutation equivariant"

print("Preds\n", preds[0,permut].cpu().numpy())
print("Permuted preds\n", perm_preds[0].cpu().numpy())

Preds
[5.4543594e-05 1.4208173e-04 6.6922468e-05 7.6413504e-05 7.7112330e-05
 8.7848457e-05 6.6820685e-05 9.9929154e-01 7.3219831e-05 6.3545609e-05]
Permuted preds
[5.4543532e-05 1.4208158e-04 6.6922395e-05 7.6413417e-05 7.7112243e-05
 8.7848362e-05 6.6820678e-05 9.9929142e-01 7.3219751e-05 6.3545544e-05]
```

You can see that the predictions are almost exactly the same, and only differ because of slight numerical differences inside the network operation.

To interpret the model a little more, we can plot the attention maps inside the model. This will give us an idea of what information the model is sharing/communicating between images, and what each head might represent. First, we need to extract the attention maps for the test batch above, and determine the discrete predictions for simplicity.

```
[38]: attention_maps = anomaly_model.get_attention_maps(inp_data, add_positional_
encoding=False)
predictions = preds.argmax(dim=-1)
```

Below we write a plot function which plots the images in the input set, the prediction of the model, and the attention maps of the different heads on layers of the transformer. Feel free to explore the attention maps for different input examples as well.

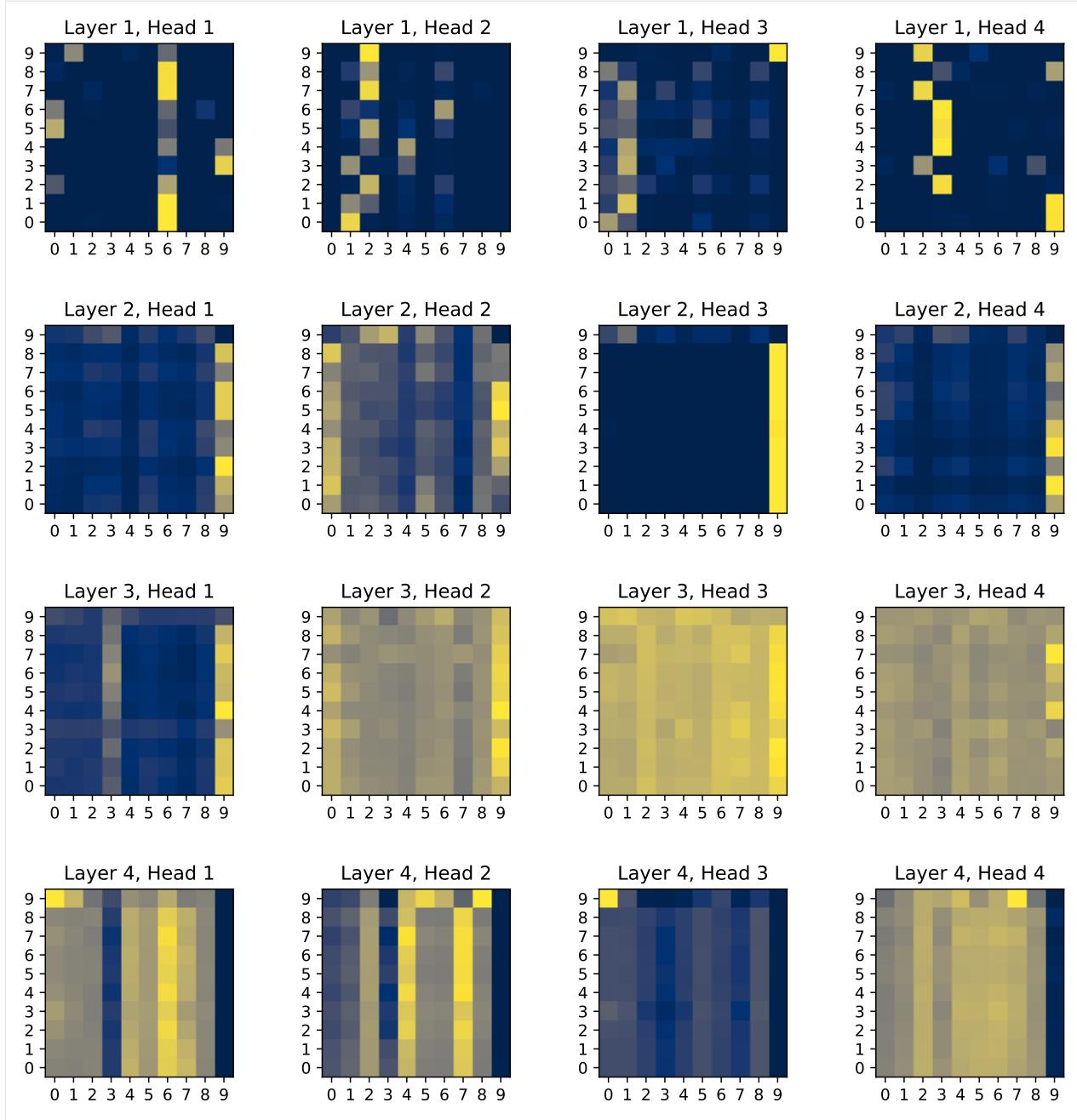
```
[39]: def visualize_prediction(idx):
    visualize_exmp(indices[idx:idx+1], test_set)
    print("Prediction:", predictions[idx].item())
    plot_attention_maps(input_data=None, attn_maps=attention_maps, idx=idx)

visualize_prediction(0)
```

Anomaly examples on CIFAR100



Prediction: 9



Depending on the random seed, you might see a slightly different input set. For the version on the website, we compare 9 tree images with a volcano. We see that multiple heads, for instance, Layer 2 Head 1, Layer 2 Head 3, and Layer 3 Head 1 focus on the last image. Additionally, the heads in Layer 4 all seem to ignore the last image and assign a very low attention probability to it. This shows that the model has indeed recognized that the image doesn't fit the setting, and hence predicted it to be the anomaly. Layer 3 Head 2-4 seems to take a slightly weighted average of all images. That might indicate that the model extracts the “average” information of all images, to compare it to the image features itself.

Let's try to find where the model actually makes a mistake. We can do this by identifying the sets where the model predicts something else than 9, as in the dataset, we ensured that the anomaly is always at the last position in the set.

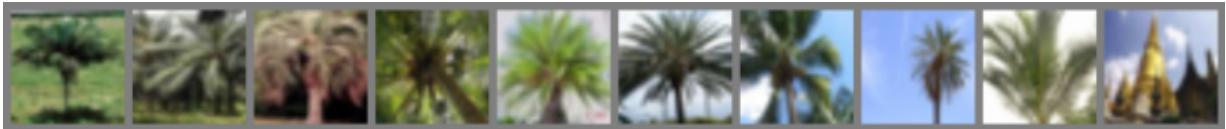
```
[40]: mistakes = torch.where(predictions != 9)[0].cpu().numpy()
print("Indices with mistake:", mistakes)

Indices with mistake: [36 49]
```

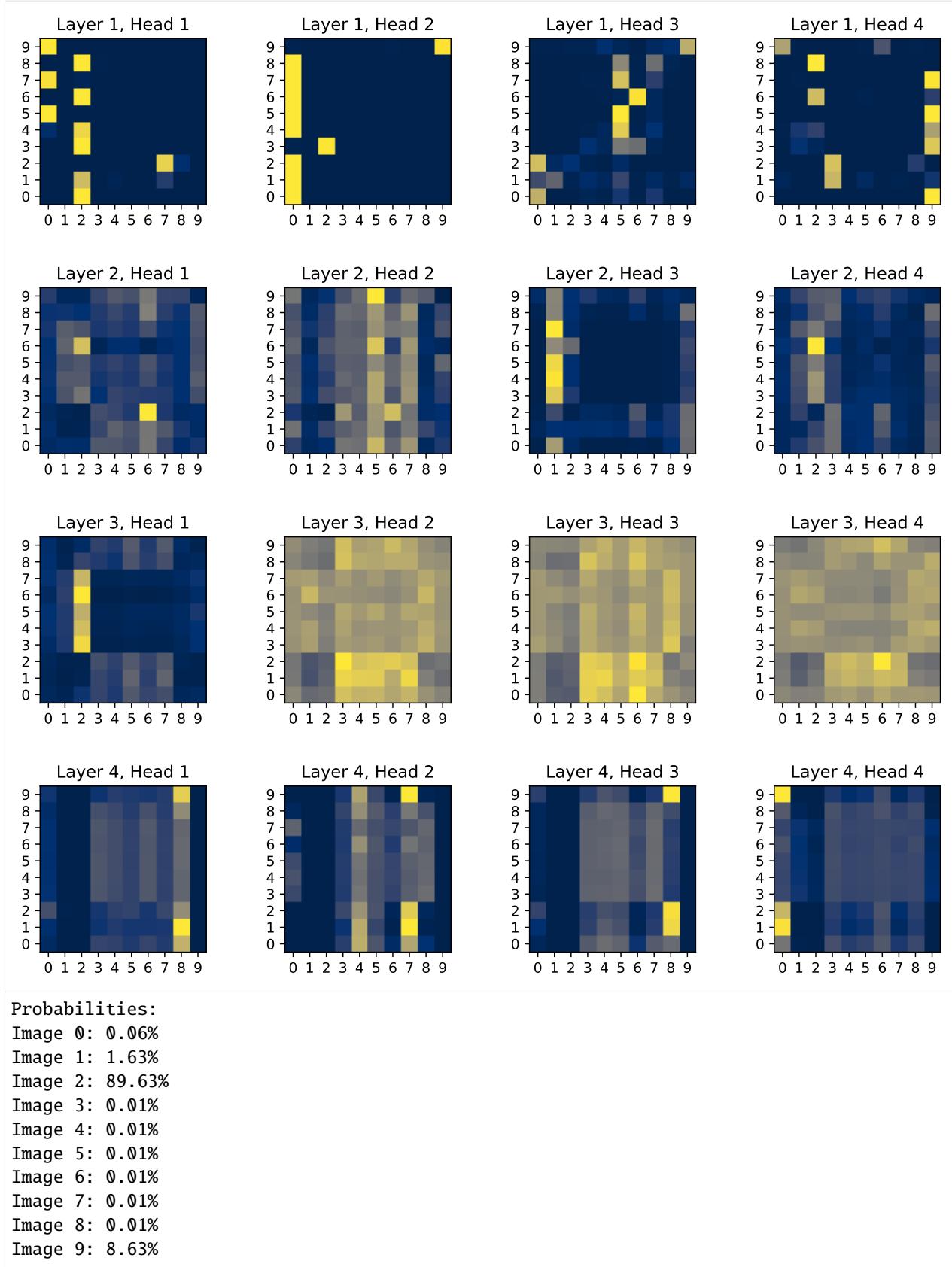
As our model achieves ~94% accuracy, we only have very little number of mistakes in a batch of 64 sets. Still, let's visualize one of them, for example the last one:

```
[41]: visualize_prediction(mistakes[-1])
print("Probabilities:")
for i, p in enumerate(preds[mistakes[-1]].cpu().numpy()):
    print(f"Image {i}: {100.0*p:.2f}%")
```

Anomaly examples on CIFAR100



Prediction: 2



In this example, the model confuses a palm tree with a building, giving a probability of ~90% to image 2, and 8% to the actual anomaly. However, the difficulty here is that the picture of the building has been taken at a similar angle as the palms. Meanwhile, image 2 shows a rather unusual palm with a different color palette, which is why the model fails here. Nevertheless, in general, the model performs quite well.

4.9.3 Conclusion

In this tutorial, we took a closer look at the Multi-Head Attention layer which uses a scaled dot product between queries and keys to find correlations and similarities between input elements. The Transformer architecture is based on the Multi-Head Attention layer and applies multiple of them in a ResNet-like block. The Transformer is a very important, recent architecture that can be applied to many tasks and datasets. Although it is best known for its success in NLP, there is so much more to it. We have seen its application on sequence-to-sequence tasks and set anomaly detection. Its property of being permutation-equivariant if we do not provide any positional encodings, allows it to generalize to many settings. Hence, it is important to know the architecture, but also its possible issues such as the gradient problem during the first iterations solved by learning rate warm-up. If you are interested in continuing with the study of the Transformer architecture, please have a look at the blog posts listed at the beginning of the tutorial notebook.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.10 Tutorial 7: Graph Neural Networks

Filled notebook:

Pre-trained models:

Recordings:

JAX+Flax version:

Author: Phillip Lippe

Note: Interested in JAX? Check out our [JAX+Flax version](#) of this tutorial!

In this tutorial, we will discuss the application of neural networks on graphs. Graph Neural Networks (GNNs) have recently gained increasing popularity in both applications and research, including domains such as social networks, knowledge graphs, recommender systems, and bioinformatics. While the theory and math behind GNNs might first seem complicated, the implementation of those models is quite simple and helps in understanding the methodology. Therefore, we will discuss the implementation of basic network layers of a GNN, namely graph convolutions, and attention layers. Finally, we will apply a GNN on a node-level, edge-level, and graph-level tasks.

Below, we will start by importing our standard libraries. We will use PyTorch Lightning as already done in Tutorial 5 and 6.

```
[1]: ## Standard libraries
import os
import json
import math
import numpy as np
import time

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()
sns.set()

## Progress bar
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
# Torchvision
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms
# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial7"

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

(continues on next page)

(continued from previous page)

```
device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print(device)

cuda:0
```

We also have a few pre-trained models we download below.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial7/"
# Files to download
pretrained_files = ["NodeLevelMLP.ckpt", "NodeLevelGNN.ckpt", "GraphLevelGraphConv.ckpt"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive folder, or contact the author with the full output including the following error:\n", e)
```

4.10.1 Graph Neural Networks

Graph representation

Before starting the discussion of specific neural network operations on graphs, we should consider how to represent a graph. Mathematically, a graph \mathcal{G} is defined as a tuple of a set of nodes/vertices V , and a set of edges/links E : $\mathcal{G} = (V, E)$. Each edge is a pair of two vertices, and represents a connection between them. For instance, let's look at the following graph:

The vertices are $V = \{1, 2, 3, 4\}$, and edges $E = \{(1, 2), (2, 3), (2, 4), (3, 4)\}$. Note that for simplicity, we assume the graph to be undirected and hence don't add mirrored pairs like $(2, 1)$. In application, vertices and edge can often have specific attributes, and edges can even be directed. The question is how we could represent this diversity in an efficient way for matrix operations. Usually, for the edges, we decide between two variants: an adjacency matrix, or a list of paired vertex indices.

The **adjacency matrix** A is a square matrix whose elements indicate whether pairs of vertices are adjacent, i.e. connected, or not. In the simplest case, A_{ij} is 1 if there is a connection from node i to j , and otherwise 0. If we have edge attributes or different categories of edges in a graph, this information can be added to the matrix as well. For an undirected graph, keep in mind that A is a symmetric matrix ($A_{ij} = A_{ji}$). For the example graph above, we have the

following adjacency matrix:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

While expressing a graph as a list of edges is more efficient in terms of memory and (possibly) computation, using an adjacency matrix is more intuitive and simpler to implement. In our implementations below, we will rely on the adjacency matrix to keep the code simple. However, common libraries use edge lists, which we will discuss later more. Alternatively, we could also use the list of edges to define a sparse adjacency matrix with which we can work as if it was a dense matrix, but allows more memory-efficient operations. PyTorch supports this with the sub-package `torch.sparse` ([documentation](#)) which is however still in a beta-stage (API might change in future).

Graph Convolutions

Graph Convolutional Networks have been introduced by Kipf et al. in 2016 at the University of Amsterdam. He also wrote a great [blog post](#) about this topic, which is recommended if you want to read about GCNs from a different perspective. GCNs are similar to convolutions in images in the sense that the “filter” parameters are typically shared over all locations in the graph. At the same time, GCNs rely on message passing methods, which means that vertices exchange information with the neighbors, and send “messages” to each other. Before looking at the math, we can try to visually understand how GCNs work. The first step is that each node creates a feature vector that represents the message it wants to send to all its neighbors. In the second step, the messages are sent to the neighbors, so that a node receives one message per adjacent node. Below we have visualized the two steps for our example graph.

If we want to formulate that in more mathematical terms, we need to first decide how to combine all the messages a node receives. As the number of messages vary across nodes, we need an operation that works for any number. Hence, the usual way to go is to sum or take the mean. Given the previous features of nodes $H^{(l)}$, the GCN layer is defined as follows:

$$H^{(l+1)} = \sigma \left(\hat{D}^{-1/2} \hat{A} \hat{D}^{-1/2} H^{(l)} W^{(l)} \right)$$

$W^{(l)}$ is the weight parameters with which we transform the input features into messages ($H^{(l)} W^{(l)}$). To the adjacency matrix A we add the identity matrix so that each node sends its own message also to itself: $\hat{A} = A + I$. Finally, to take the average instead of summing, we calculate the matrix \hat{D} which is a diagonal matrix with D_{ii} denoting the number of neighbors node i has. σ represents an arbitrary activation function, and not necessarily the sigmoid (usually a ReLU-based activation function is used in GNNs).

When implementing the GCN layer in PyTorch, we can take advantage of the flexible operations on tensors. Instead of defining a matrix \hat{D} , we can simply divide the summed messages by the number of neighbors afterward. Additionally, we replace the weight matrix with a linear layer, which additionally allows us to add a bias. Written as a PyTorch module, the GCN layer is defined as follows:

```
[3]: class GCNLayer(nn.Module):

    def __init__(self, c_in, c_out):
        super().__init__()
        self.projection = nn.Linear(c_in, c_out)

    def forward(self, node_feats, adj_matrix):
        """
        Inputs:
            node_feats - Tensor with node features of shape [batch_size, num_nodes, c_in]
```

(continues on next page)

(continued from previous page)

```

adj_matrix - Batch of adjacency matrices of the graph. If there is an edge
from i to j, adj_matrix[b,i,j]=1 else 0.
Supports directed edges by non-symmetric matrices. Assumes to
already have added the identity connections.
Shape: [batch_size, num_nodes, num_nodes]
"""

# Num neighbours = number of incoming edges
num_neighbours = adj_matrix.sum(dim=-1, keepdims=True)
node_feats = self.projection(node_feats)
node_feats = torch.bmm(adj_matrix, node_feats)
node_feats = node_feats / num_neighbours
return node_feats

```

To further understand the GCN layer, we can apply it to our example graph above. First, let's specify some node features and the adjacency matrix with added self-connections:

```
[4]: node_feats = torch.arange(8, dtype=torch.float32).view(1, 4, 2)
adj_matrix = torch.Tensor([[[1, 1, 0, 0],
                           [1, 1, 1, 1],
                           [0, 1, 1, 1],
                           [0, 1, 1, 1]]])

print("Node features:\n", node_feats)
print("\nAdjacency matrix:\n", adj_matrix)

Node features:
tensor([[[0., 1.],
         [2., 3.],
         [4., 5.],
         [6., 7.]]])

Adjacency matrix:
tensor([[[1., 1., 0., 0.],
         [1., 1., 1., 1.],
         [0., 1., 1., 1.],
         [0., 1., 1., 1.]]])
```

Next, let's apply a GCN layer to it. For simplicity, we initialize the linear weight matrix as an identity matrix so that the input features are equal to the messages. This makes it easier for us to verify the message passing operation.

```
[5]: layer = GCNLayer(c_in=2, c_out=2)
layer.projection.weight.data = torch.Tensor([[1., 0.], [0., 1.]])
layer.projection.bias.data = torch.Tensor([0., 0.])

with torch.no_grad():
    out_feats = layer(node_feats, adj_matrix)

print("Adjacency matrix", adj_matrix)
print("Input features", node_feats)
print("Output features", out_feats)

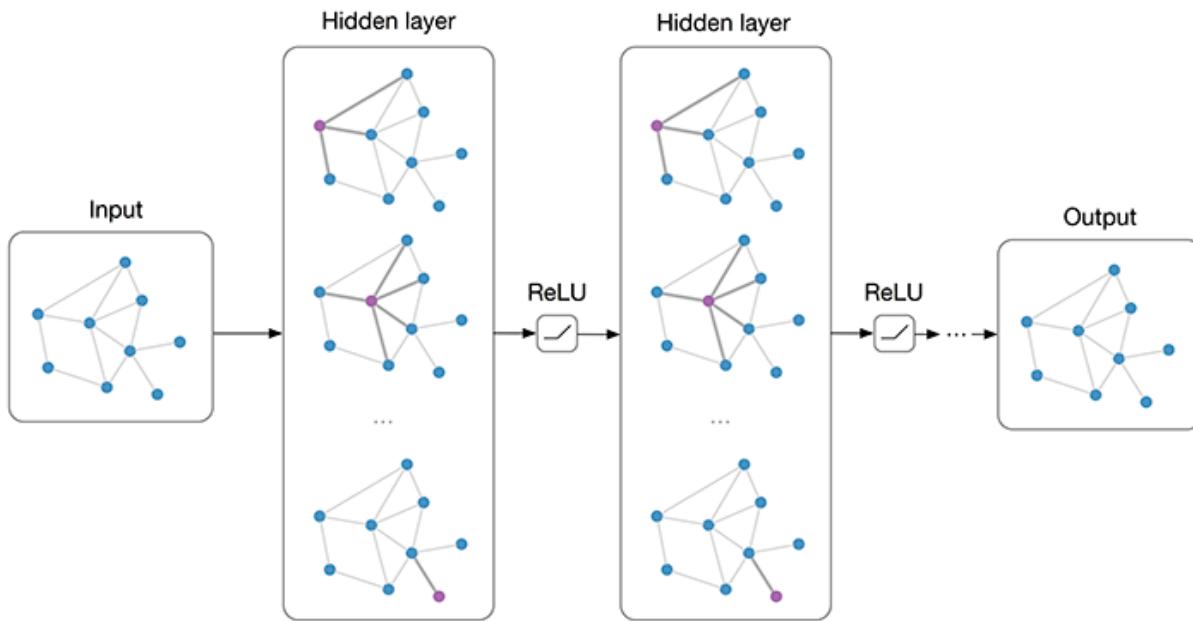
Adjacency matrix tensor([[[1., 1., 0., 0.],
                           [1., 1., 1., 1.]],
```

(continues on next page)

(continued from previous page)

```
[0., 1., 1., 1.],
[0., 1., 1., 1.]]])
Input features tensor([[[0., 1.],
[2., 3.],
[4., 5.],
[6., 7.]]])
Output features tensor([[[1., 2.],
[3., 4.],
[4., 5.],
[4., 5.]]])
```

As we can see, the first node's output values are the average of itself and the second node. Similarly, we can verify all other nodes. However, in a GNN, we would also want to allow feature exchange between nodes beyond its neighbors. This can be achieved by applying multiple GCN layers, which gives us the final layout of a GNN. The GNN can be built up by a sequence of GCN layers and non-linearities such as ReLU. For a visualization, see below (figure credit - Thomas Kipf, 2016).



However, one issue we can see from looking at the example above is that the output features for nodes 3 and 4 are the same because they have the same adjacent nodes (including itself). Therefore, GCN layers can make the network forget node-specific information if we just take a mean over all messages. Multiple possible improvements have been proposed. While the simplest option might be using residual connections, the more common approach is to either weigh the self-connections higher or define a separate weight matrix for the self-connections. Alternatively, we can re-visit a concept from the last tutorial: attention.

Graph Attention

If you remember from the last tutorial, attention describes a weighted average of multiple elements with the weights dynamically computed based on an input query and elements' keys (if you haven't read Tutorial 6 yet, it is recommended to at least go through the very first section called [What is Attention?](#)). This concept can be similarly applied to graphs, one of such is the Graph Attention Network (called GAT, proposed by [Velickovic et al., 2017](#)). Similarly to the GCN, the graph attention layer creates a message for each node using a linear layer/weight matrix. For the attention part, it uses the message from the node itself as a query, and the messages to average as both keys and values (note that this also includes the message to itself). The score function f_{attn} is implemented as a one-layer MLP which maps the query and key to a single value. The MLP looks as follows (figure credit - [Velickovic et al.](#)):

h_i and h_j are the original features from node i and j respectively, and represent the messages of the layer with \mathbf{W} as weight matrix. \mathbf{a} is the weight matrix of the MLP, which has the shape $[1, 2 \times d_{\text{message}}]$, and α_{ij} the final attention weight from node i to j . The calculation can be described as follows:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_j]))}{\sum_{k \in \mathcal{N}_i} \exp(\text{LeakyReLU}(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_k]))}$$

The operator $||$ represents the concatenation, and \mathcal{N}_i the indices of the neighbors of node i . Note that in contrast to usual practice, we apply a non-linearity (here LeakyReLU) before the softmax over elements. Although it seems like a minor change at first, it is crucial for the attention to depend on the original input. Specifically, let's remove the non-linearity for a second, and try to simplify the expression:

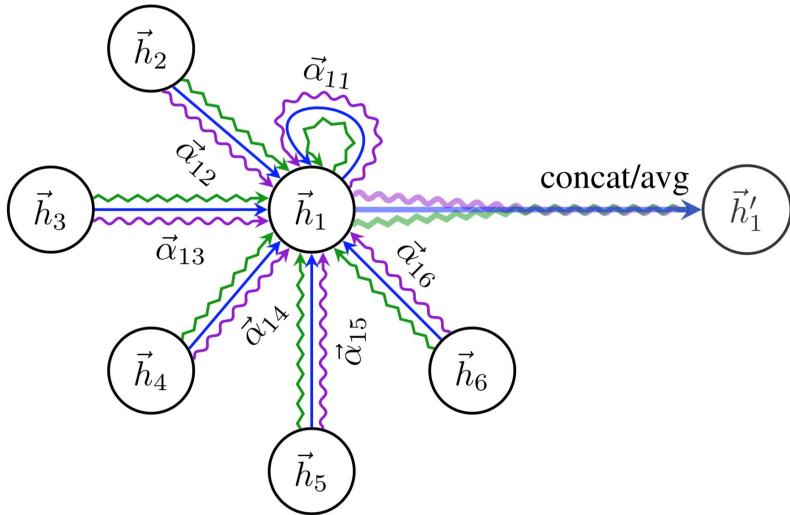
$$\begin{aligned} \alpha_{ij} &= \frac{\exp(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_j])}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_k])} \\ &= \frac{\exp(\mathbf{a}_{:,d/2}\mathbf{W}h_i + \mathbf{a}_{:,d/2}\mathbf{W}h_j)}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}_{:,d/2}\mathbf{W}h_i + \mathbf{a}_{:,d/2}\mathbf{W}h_k)} \\ &= \frac{\exp(\mathbf{a}_{:,d/2}\mathbf{W}h_i) \cdot \exp(\mathbf{a}_{:,d/2}\mathbf{W}h_j)}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}_{:,d/2}\mathbf{W}h_i) \cdot \exp(\mathbf{a}_{:,d/2}\mathbf{W}h_k)} \\ &= \frac{\exp(\mathbf{a}_{:,d/2}\mathbf{W}h_j)}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}_{:,d/2}\mathbf{W}h_k)} \end{aligned}$$

We can see that without the non-linearity, the attention term with h_i actually cancels itself out, resulting in the attention being independent of the node itself. Hence, we would have the same issue as the GCN of creating the same output features for nodes with the same neighbors. This is why the LeakyReLU is crucial and adds some dependency on h_i to the attention.

Once we obtain all attention factors, we can calculate the output features for each node by performing the weighted average:

$$h'_i = \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}h_j \right)$$

σ is yet another non-linearity, as in the GCN layer. Visually, we can represent the full message passing in an attention layer as follows (figure credit - [Velickovic et al.](#)):



To increase the expressiveness of the graph attention network, Velickovic et al. proposed to extend it to multiple heads similar to the Multi-Head Attention block in Transformers. This results in N attention layers being applied in parallel. In the image above, it is visualized as three different colors of arrows (green, blue, and purple) that are afterward concatenated. The average is only applied for the very final prediction layer in a network.

After having discussed the graph attention layer in detail, we can implement it below:

```
[6]: class GATLayer(nn.Module):

    def __init__(self, c_in, c_out, num_heads=1, concat_heads=True, alpha=0.2):
        """
        Inputs:
            c_in - Dimensionality of input features
            c_out - Dimensionality of output features
            num_heads - Number of heads, i.e. attention mechanisms to apply in parallel.
        The
            output features are equally split up over the heads if concat_
        heads=True.
            concat_heads - If True, the output of the different heads is concatenated_
        instead of averaged.
            alpha - Negative slope of the LeakyReLU activation.
        """
        super().__init__()
        self.num_heads = num_heads
        self.concat_heads = concat_heads
        if self.concat_heads:
            assert c_out % num_heads == 0, "Number of output features must be a multiple_
        of the count of heads."
            c_out = c_out // num_heads

        # Sub-modules and parameters needed in the layer
        self.projection = nn.Linear(c_in, c_out * num_heads)
        self.a = nn.Parameter(torch.Tensor(num_heads, 2 * c_out)) # One per head
        self.leakyrelu = nn.LeakyReLU(alpha)

        # Initialization from the original implementation
        nn.init.xavier_uniform_(self.projection.weight.data, gain=1.414)
```

(continues on next page)

(continued from previous page)

```

nn.init.xavier_uniform_(self.a.data, gain=1.414)

def forward(self, node_feats, adj_matrix, print_attn_probs=False):
    """
    Inputs:
        node_feats - Input features of the node. Shape: [batch_size, c_in]
        adj_matrix - Adjacency matrix including self-connections. Shape: [batch_size,
    ↵ num_nodes, num_nodes]
        print_attn_probs - If True, the attention weights are printed during the
    ↵ forward pass (for debugging purposes)
    """
    batch_size, num_nodes = node_feats.size(0), node_feats.size(1)

    # Apply linear layer and sort nodes by head
    node_feats = self.projection(node_feats)
    node_feats = node_feats.view(batch_size, num_nodes, self.num_heads, -1)

    # We need to calculate the attention logits for every edge in the adjacency_
    ↵matrix
    # Doing this on all possible combinations of nodes is very expensive
    # => Create a tensor of [W*h_i||W*h_j] with i and j being the indices of all_
    ↵edges
    edges = adj_matrix.nonzero(as_tuple=False) # Returns indices where the adjacency_
    ↵matrix is not 0 => edges
    node_feats_flat = node_feats.view(batch_size * num_nodes, self.num_heads, -1)
    edge_indices_row = edges[:, 0] * num_nodes + edges[:, 1]
    edge_indices_col = edges[:, 0] * num_nodes + edges[:, 2]
    a_input = torch.cat([
        torch.index_select(input=node_feats_flat, index=edge_indices_row, dim=0),
        torch.index_select(input=node_feats_flat, index=edge_indices_col, dim=0)
    ], dim=-1) # Index select returns a tensor with node_feats_flat being indexed at_
    ↵the desired positions along dim=0

    # Calculate attention MLP output (independent for each head)
    attn_logits = torch.einsum('bhc,hc->bh', a_input, self.a)
    attn_logits = self.leakyrelu(attn_logits)

    # Map list of attention values back into a matrix
    attn_matrix = attn_logits.new_zeros(adj_matrix.shape+(self.num_heads,)).fill_(-
    ↵9e15)
    attn_matrix[adj_matrix[...,None].repeat(1,1,1,self.num_heads) == 1] = attn_
    ↵logits.reshape(-1)

    # Weighted average of attention
    attn_probs = F.softmax(attn_matrix, dim=2)
    if print_attn_probs:
        print("Attention probs\n", attn_probs.permute(0, 3, 1, 2))
    node_feats = torch.einsum('bijh,bjhc->bihc', attn_probs, node_feats)

    # If heads should be concatenated, we can do this by reshaping. Otherwise, take_
    ↵mean
    if self.concat_heads:

```

(continues on next page)

(continued from previous page)

```

    node_feats = node_feats.reshape(batch_size, num_nodes, -1)
else:
    node_feats = node_feats.mean(dim=2)

return node_feats

```

Again, we can apply the graph attention layer on our example graph above to understand the dynamics better. As before, the input layer is initialized as an identity matrix, but we set \mathbf{a} to be a vector of arbitrary numbers to obtain different attention values. We use two heads to show the parallel, independent attention mechanisms working in the layer.

```
[7]: layer = GATLayer(2, 2, num_heads=2)
layer.projection.weight.data = torch.Tensor([[1., 0.], [0., 1.]])
layer.projection.bias.data = torch.Tensor([0., 0.])
layer.a.data = torch.Tensor([[-0.2, 0.3], [0.1, -0.1]])

with torch.no_grad():
    out_feats = layer(node_feats, adj_matrix, print_attn_probs=True)

print("Adjacency matrix", adj_matrix)
print("Input features", node_feats)
print("Output features", out_feats)

Attention probs
tensor([[[[0.3543, 0.6457, 0.0000, 0.0000],
          [0.1096, 0.1450, 0.2642, 0.4813],
          [0.0000, 0.1858, 0.2885, 0.5257],
          [0.0000, 0.2391, 0.2696, 0.4913]],

         [[0.5100, 0.4900, 0.0000, 0.0000],
          [0.2975, 0.2436, 0.2340, 0.2249],
          [0.0000, 0.3838, 0.3142, 0.3019],
          [0.0000, 0.4018, 0.3289, 0.2693]]]]]
Adjacency matrix tensor([[[[1., 1., 0., 0.],
                           [1., 1., 1., 1.],
                           [0., 1., 1., 1.],
                           [0., 1., 1., 1.]]]])
Input features tensor([[[[0., 1.],
                        [2., 3.],
                        [4., 5.],
                        [6., 7.]]]])
Output features tensor([[[[1.2913, 1.9800],
                         [4.2344, 3.7725],
                         [4.6798, 4.8362],
                         [4.5043, 4.7351]]]])
```

We recommend that you try to calculate the attention matrix at least for one head and one node for yourself. The entries are 0 where there does not exist an edge between i and j . For the others, we see a diverse set of attention probabilities. Moreover, the output features of node 3 and 4 are now different although they have the same neighbors.

4.10.2 PyTorch Geometric

We had mentioned before that implementing graph networks with adjacency matrix is simple and straight-forward but can be computationally expensive for large graphs. Many real-world graphs can reach over 200k nodes, for which adjacency matrix-based implementations fail. There are a lot of optimizations possible when implementing GNNs, and luckily, there exist packages that provide such layers. The most popular packages for PyTorch are [PyTorch Geometric](#) and the [Deep Graph Library](#) (the latter being actually framework agnostic). Which one to use depends on the project you are planning to do and personal taste. In this tutorial, we will look at PyTorch Geometric as part of the PyTorch family. Similar to PyTorch Lightning, PyTorch Geometric is not installed by default on GoogleColab (and actually also not in our d12021 environment due to many dependencies that would be unnecessary for the practicals). Hence, let's import and/or install it below:

```
[8]: # torch geometric
try:
    import torch_geometric
except ModuleNotFoundError:
    # Installing torch geometric packages with specific CUDA+PyTorch version.
    # See https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html for
    # details
    TORCH = torch.__version__.split('+')[0]
    CUDA = 'cu' + torch.version.cuda.replace('.', '')

    !pip install torch-scatter      -f https://pytorch-geometric.com/whl/torch-{TORCH}+
    ↪{CUDA}.html
    !pip install torch-sparse     -f https://pytorch-geometric.com/whl/torch-{TORCH}+
    ↪{CUDA}.html
    !pip install torch-cluster    -f https://pytorch-geometric.com/whl/torch-{TORCH}+
    ↪{CUDA}.html
    !pip install torch-spline-conv -f https://pytorch-geometric.com/whl/torch-{TORCH}+
    ↪{CUDA}.html
    !pip install torch-geometric
    import torch_geometric
import torch_geometric.nn as geom_nn
import torch_geometric.data as geom_data
```

RDKit WARNING: [19:12:50] Enabling RDKit 2019.09.3 jupyter extensions

PyTorch Geometric provides us a set of common graph layers, including the GCN and GAT layer we implemented above. Additionally, similar to PyTorch's torchvision, it provides the common graph datasets and transformations on those to simplify training. Compared to our implementation above, PyTorch Geometric uses a list of index pairs to represent the edges. The details of this library will be explored further in our experiments.

In our tasks below, we want to allow us to pick from a multitude of graph layers. Thus, we define again below a dictionary to access those using a string:

```
[9]: gnn_layer_by_name = {
    "GCN": geom_nn.GCNConv,
    "GAT": geom_nn.GATConv,
    "GraphConv": geom_nn.GraphConv
}
```

Additionally to GCN and GAT, we added the layer `geom_nn.GraphConv` ([documentation](#)). GraphConv is a GCN with a separate weight matrix for the self-connections. Mathematically, this would be:

$$\mathbf{x}_i^{(l+1)} = \mathbf{W}_1^{(l+1)} \mathbf{x}_i^{(l)} + \mathbf{W}_2^{(l+1)} \sum_{j \in \mathcal{N}_i} \mathbf{x}_j^{(l)}$$

In this formula, the neighbor's messages are added instead of averaged. However, PyTorch Geometric provides the argument `aggr` to switch between summing, averaging, and max pooling.

4.10.3 Experiments on graph structures

Tasks on graph-structured data can be grouped into three groups: node-level, edge-level and graph-level. The different levels describe on which level we want to perform classification/regression. We will discuss all three types in more detail below.

Node-level tasks: Semi-supervised node classification

Node-level tasks have the goal to classify nodes in a graph. Usually, we have given a single, large graph with >1000 nodes of which a certain amount of nodes are labeled. We learn to classify those labeled examples during training and try to generalize to the unlabeled nodes.

A popular example that we will use in this tutorial is the Cora dataset, a citation network among papers. The Cora consists of 2708 scientific publications with links between each other representing the citation of one paper by another. The task is to classify each publication into one of seven classes. Each publication is represented by a bag-of-words vector. This means that we have a vector of 1433 elements for each publication, where a 1 at feature i indicates that the i -th word of a pre-defined dictionary is in the article. Binary bag-of-words representations are commonly used when we need very simple encodings, and already have an intuition of what words to expect in a network. There exist much better approaches, but we will leave this to the NLP courses to discuss.

We will load the dataset below:

```
[10]: cora_dataset = torch_geometric.datasets.Planetoid(root=DATASET_PATH, name="Cora")
```

Let's look at how PyTorch Geometric represents the graph data. Note that although we have a single graph, PyTorch Geometric returns a dataset for compatibility to other datasets.

```
[11]: cora_dataset[0]
```

```
[11]: Data(edge_index=[2, 10556], test_mask=[2708], train_mask=[2708], val_mask=[2708],  
        x=[2708, 1433], y=[2708])
```

The graph is represented by a `Data` object ([documentation](#)) which we can access as a standard Python namespace. The edge index tensor is the list of edges in the graph and contains the mirrored version of each edge for undirected graphs. The `train_mask`, `val_mask`, and `test_mask` are boolean masks that indicate which nodes we should use for training, validation, and testing. The `x` tensor is the feature tensor of our 2708 publications, and `y` the labels for all nodes.

After having seen the data, we can implement a simple graph neural network. The GNN applies a sequence of graph layers (GCN, GAT, or GraphConv), ReLU as activation function, and dropout for regularization. See below for the specific implementation.

```
[12]: class GNNModel(nn.Module):
```

```
    def __init__(self, c_in, c_hidden, c_out, num_layers=2, layer_name="GCN", dp_rate=0.  
                 1, **kwargs):  
        """  
        Inputs:  
            c_in - Dimension of input features  
            c_hidden - Dimension of hidden features  
            c_out - Dimension of the output features. Usually number of classes in  
                    classification
```

(continues on next page)

(continued from previous page)

```

    num_layers - Number of "hidden" graph layers
    layer_name - String of the graph layer to use
    dp_rate - Dropout rate to apply throughout the network
    kwargs - Additional arguments for the graph layer (e.g. number of heads for
→ GAT)
    """
    super().__init__()
    gnn_layer = gnn_layer_by_name[layer_name]

    layers = []
    in_channels, out_channels = c_in, c_hidden
    for l_idx in range(num_layers-1):
        layers += [
            gnn_layer(in_channels=in_channels,
                      out_channels=out_channels,
                      **kwargs),
            nn.ReLU(inplace=True),
            nn.Dropout(dp_rate)
        ]
        in_channels = c_hidden
    layers += [gnn_layer(in_channels=in_channels,
                         out_channels=c_out,
                         **kwargs)]
    self.layers = nn.ModuleList(layers)

    def forward(self, x, edge_index):
        """
        Inputs:
            x - Input features per node
            edge_index - List of vertex index pairs representing the edges in the graph
→ (PyTorch geometric notation)
        """
        for l in self.layers:
            # For graph layers, we need to add the "edge_index" tensor as additional
→ input
            # All PyTorch Geometric graph layer inherit the class "MessagePassing", hence
            # we can simply check the class type.
            if isinstance(l, geom_nn.MessagePassing):
                x = l(x, edge_index)
            else:
                x = l(x)
        return x

```

Good practice in node-level tasks is to create an MLP baseline that is applied to each node independently. This way we can verify whether adding the graph information to the model indeed improves the prediction, or not. It might also be that the features per node are already expressive enough to clearly point towards a specific class. To check this, we implement a simple MLP below.

```
[13]: class MLPModel(nn.Module):

    def __init__(self, c_in, c_hidden, c_out, num_layers=2, dp_rate=0.1):
        """

```

(continues on next page)

(continued from previous page)

```

Inputs:
    c_in - Dimension of input features
    c_hidden - Dimension of hidden features
    c_out - Dimension of the output features. Usually number of classes in classification
    num_layers - Number of hidden layers
    dp_rate - Dropout rate to apply throughout the network
    """
super().__init__()
layers = []
in_channels, out_channels = c_in, c_hidden
for l_idx in range(num_layers-1):
    layers += [
        nn.Linear(in_channels, out_channels),
        nn.ReLU(inplace=True),
        nn.Dropout(dp_rate)
    ]
    in_channels = c_hidden
layers += [nn.Linear(in_channels, c_out)]
self.layers = nn.Sequential(*layers)

def forward(self, x, *args, **kwargs):
    """
    Inputs:
        x - Input features per node
    """
    return self.layers(x)

```

Finally, we can merge the models into a PyTorch Lightning module which handles the training, validation, and testing for us.

```
[14]: class NodeLevelGNN(pl.LightningModule):

    def __init__(self, model_name, **model_kwargs):
        super().__init__()
        # Saving hyperparameters
        self.save_hyperparameters()

        if model_name == "MLP":
            self.model = MLPModel(**model_kwargs)
        else:
            self.model = GNNModel(**model_kwargs)
        self.loss_module = nn.CrossEntropyLoss()

    def forward(self, data, mode="train"):
        x, edge_index = data.x, data.edge_index
        x = self.model(x, edge_index)

        # Only calculate the loss on the nodes corresponding to the mask
        if mode == "train":
            mask = data.train_mask
        elif mode == "val":
```

(continues on next page)

(continued from previous page)

```

        mask = data.val_mask
    elif mode == "test":
        mask = data.test_mask
    else:
        assert False, f"Unknown forward mode: {mode}"

    loss = self.loss_module(x[mask], data.y[mask])
    acc = (x[mask].argmax(dim=-1) == data.y[mask]).sum().float() / mask.sum()
    return loss, acc

def configure_optimizers(self):
    # We use SGD here, but Adam works as well
    optimizer = optim.SGD(self.parameters(), lr=0.1, momentum=0.9, weight_decay=2e-3)
    return optimizer

def training_step(self, batch, batch_idx):
    loss, acc = self.forward(batch, mode="train")
    self.log('train_loss', loss)
    self.log('train_acc', acc)
    return loss

def validation_step(self, batch, batch_idx):
    _, acc = self.forward(batch, mode="val")
    self.log('val_acc', acc)

def test_step(self, batch, batch_idx):
    _, acc = self.forward(batch, mode="test")
    self.log('test_acc', acc)

```

Additionally to the Lightning module, we define a training function below. As we have a single graph, we use a batch size of 1 for the data loader and share the same data loader for the train, validation, and test set (the mask is picked inside the Lightning module). Besides, we set the argument `enable_progress_bar` to `False` as it usually shows the progress per epoch, but an epoch only consists of a single step. The rest of the code is very similar to what we have seen in Tutorial 5 and 6 already.

```
[15]: def train_node_classifier(model_name, dataset, **model_kwargs):
    pl.seed_everything(42)
    node_data_loader = geom_data.DataLoader(dataset, batch_size=1)

    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "NodeLevel" + model_name)
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                          callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
                                                      monitor="val_acc")],
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=200,
                          enable_progress_bar=False) # False because epoch size is 1
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need
```

(continues on next page)

(continued from previous page)

```

# Check whether pretrained model exists. If yes, load it and skip training
pretrained_filename = os.path.join(CHECKPOINT_PATH, f"NodeLevel{model_name}.ckpt")
if os.path.isfile(pretrained_filename):
    print("Found pretrained model, loading...")
    model = NodeLevelGNN.load_from_checkpoint(pretrained_filename)
else:
    pl.seed_everything()
    model = NodeLevelGNN(model_name=model_name, c_in=dataset.num_node_features, c_
→out=dataset.num_classes, **model_kwargs)
    trainer.fit(model, node_data_loader, node_data_loader)
    model = NodeLevelGNN.load_from_checkpoint(trainer.checkpoint_callback.best_model_
→path)

# Test best model on the test set
test_result = trainer.test(model, node_data_loader, verbose=False)
batch = next(iter(node_data_loader))
batch = batch.to(model.device)
_, train_acc = model.forward(batch, mode="train")
_, val_acc = model.forward(batch, mode="val")
result = {"train": train_acc,
          "val": val_acc,
          "test": test_result[0]['test_acc']}
return model, result

```

Finally, we can train our models. First, let's train the simple MLP:

```
[16]: # Small function for printing the test scores
def print_results(result_dict):
    if "train" in result_dict:
        print(f"Train accuracy: {(100.0*result_dict['train']):.2f}%")
    if "val" in result_dict:
        print(f"Val accuracy: {(100.0*result_dict['val']):.2f}%")
    print(f"Test accuracy: {(100.0*result_dict['test']):.2f}%")
```

```
[17]: node_mlp_model, node_mlp_result = train_node_classifier(model_name="MLP",
                                                               dataset=cora_dataset,
                                                               c_hidden=16,
                                                               num_layers=2,
                                                               dp_rate=0.1)

print_results(node_mlp_result)

GPU available: True, used: True
WARNING: Logging before flag parsing goes to stderr.
I1113 19:12:52.401648 139969460983616 distributed.py:49] GPU available: True, used: True
TPU available: False, using: 0 TPU cores
I1113 19:12:52.403518 139969460983616 distributed.py:49] TPU available: False, using: 0_
→TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
I1113 19:12:52.404974 139969460983616 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_
→VISIBLE_DEVICES: [0]
```

```
Found pretrained model, loading...
Train accuracy: 96.43%
Val accuracy: 52.60%
Test accuracy: 60.60%

/home/phillip/anaconda3/envs/nlp1/lib/python3.7/site-packages/pytorch_lightning/
↳ utilities/distributed.py:45: UserWarning: The dataloader, test dataloader 0, does not
↳ have many workers which may be a bottleneck. Consider increasing the value of the `num_
↳ workers` argument` (try 16 which is the number of cpus on this machine) in the
↳ `DataLoader` init to improve performance.
    warnings.warn(*args, **kwargs)
```

Although the MLP can overfit on the training dataset because of the high-dimensional input features, it does not perform too well on the test set. Let's see if we can beat this score with our graph networks:

```
[18]: node_gnn_model, node_gnn_result = train_node_classifier(model_name="GNN",
                                                               layer_name="GCN",
                                                               dataset=cora_dataset,
                                                               c_hidden=16,
                                                               num_layers=2,
                                                               dp_rate=0.1)

print_results(node_gnn_result)

GPU available: True, used: True
I1113 19:12:53.906714 139969460983616 distributed.py:49] GPU available: True, used: True
TPU available: False, using: 0 TPU cores
I1113 19:12:53.907762 139969460983616 distributed.py:49] TPU available: False, using: 0
↳ TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
I1113 19:12:53.909639 139969460983616 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_
↳ VISIBLE_DEVICES: [0]

Found pretrained model, loading...
Train accuracy: 100.00%
Val accuracy: 77.20%
Test accuracy: 82.40%
```

As we would have hoped for, the GNN model outperforms the MLP by quite a margin. This shows that using the graph information indeed improves our predictions and lets us generalize better.

The hyperparameters in the model have been chosen to create a relatively small network. This is because the first layer with an input dimension of 1433 can be relatively expensive to perform for large graphs. In general, GNNs can become relatively expensive for very big graphs. This is why such GNNs either have a small hidden size or use a special batching strategy where we sample a connected subgraph of the big, original graph.

Edge-level tasks: Link prediction

In some applications, we might have to predict on an edge-level instead of node-level. The most common edge-level task in GNN is link prediction. Link prediction means that given a graph, we want to predict whether there will be/should be an edge between two nodes or not. For example, in a social network, this is used by Facebook and co to propose new friends to you. Again, graph level information can be crucial to perform this task. The output prediction is usually done by performing a similarity metric on the pair of node features, which should be 1 if there should be a link, and otherwise close to 0. To keep the tutorial short, we will not implement this task ourselves. Nevertheless, there are many good resources out there if you are interested in looking closer at this task. Tutorials and papers for this topic include:

- PyTorch Geometric example
- Graph Neural Networks: A Review of Methods and Applications, Zhou et al. 2019
- Link Prediction Based on Graph Neural Networks, Zhang and Chen, 2018.

Graph-level tasks: Graph classification

Finally, in this part of the tutorial, we will have a closer look at how to apply GNNs to the task of graph classification. The goal is to classify an entire graph instead of single nodes or edges. Therefore, we are also given a dataset of multiple graphs that we need to classify based on some structural graph properties. The most common task for graph classification is molecular property prediction, in which molecules are represented as graphs. Each atom is linked to a node, and edges in the graph are the bonds between atoms. For example, look at the figure below.

On the left, we have an arbitrary, small molecule with different atoms, whereas the right part of the image shows the graph representation. The atom types are abstracted as node features (e.g. a one-hot vector), and the different bond types are used as edge features. For simplicity, we will neglect the edge attributes in this tutorial, but you can include by using methods like the [Relational Graph Convolution](#) that uses a different weight matrix for each edge type.

The dataset we will use below is called the MUTAG dataset. It is a common small benchmark for graph classification algorithms, and contain 188 graphs with 18 nodes and 20 edges on average for each graph. The graph nodes have 7 different labels/atom types, and the binary graph labels represent “their mutagenic effect on a specific gram negative bacterium” (the specific meaning of the labels are not too important here). The dataset is part of a large collection of different graph classification datasets, known as the [TUDatasets](#), which is directly accessible via `torch_geometric.datasets.TUDataset` ([documentation](#)) in PyTorch Geometric. We can load the dataset below.

```
[19]: tu_dataset = torch_geometric.datasets.TUDataset(root=DATASET_PATH, name="MUTAG")
```

Let's look at some statistics for the dataset:

```
[20]: print("Data object:", tu_dataset.data)
print("Length:", len(tu_dataset))
print(f"Average label: {tu_dataset.data.y.float().mean().item():.2f}")

Data object: Data(edge_attr=[7442, 4], edge_index=[2, 7442], x=[3371, 7], y=[188])
Length: 188
Average label: 0.66
```

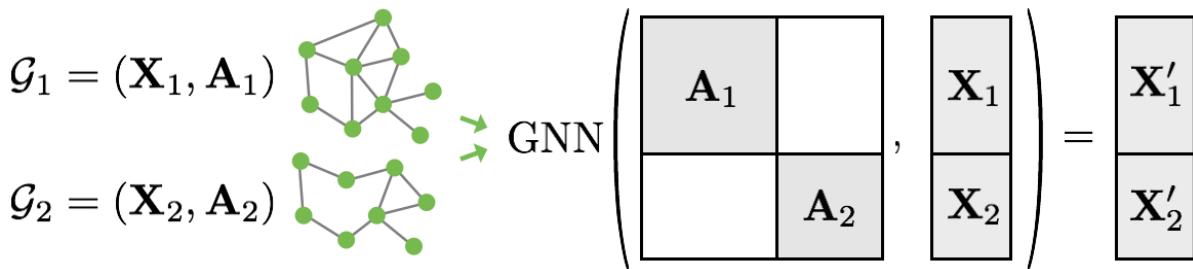
The first line shows how the dataset stores different graphs. The nodes, edges, and labels of each graph are concatenated to one tensor, and the dataset stores the indices where to split the tensors correspondingly. The length of the dataset is the number of graphs we have, and the “average label” denotes the percentage of the graph with label 1. As long as the percentage is in the range of 0.5, we have a relatively balanced dataset. It happens quite often that graph datasets are very imbalanced, hence checking the class balance is always a good thing to do.

Next, we will split our dataset into a training and test part. Note that we do not use a validation set this time because of the small size of the dataset. Therefore, our model might overfit slightly on the validation set due to the noise of the evaluation, but we still get an estimate of the performance on untrained data.

```
[21]: torch.manual_seed(42)
tu_dataset.shuffle()
train_dataset = tu_dataset[:150]
test_dataset = tu_dataset[150:]
```

When using a data loader, we encounter a problem with batching N graphs. Each graph in the batch can have a different number of nodes and edges, and hence we would require a lot of padding to obtain a single tensor. Torch geometric

uses a different, more efficient approach: we can view the N graphs in a batch as a single large graph with concatenated node and edge list. As there is no edge between the N graphs, running GNN layers on the large graph gives us the same output as running the GNN on each graph separately. Visually, this batching strategy is visualized below (figure credit - PyTorch Geometric team, [tutorial here](#)).



The adjacency matrix is zero for any nodes that come from two different graphs, and otherwise according to the adjacency matrix of the individual graph. Luckily, this strategy is already implemented in torch geometric, and hence we can use the corresponding data loader:

```
[22]: graph_train_loader = geom_data.DataLoader(train_dataset, batch_size=64, shuffle=True)
graph_val_loader = geom_data.DataLoader(test_dataset, batch_size=64) # Additional loader
    ↪ if you want to change to a larger dataset
graph_test_loader = geom_data.DataLoader(test_dataset, batch_size=64)
```

Let's load a batch below to see the batching in action:

We have 38 graphs stacked together for the test dataset. The batch indices, stored in `batch`, show that the first 12 nodes belong to the first graph, the next 22 to the second graph, and so on. These indices are important for performing the final prediction. To perform a prediction over a whole graph, we usually perform a pooling operation over all nodes after running the GNN model. In this case, we will use the average pooling. Hence, we need to know which nodes should be included in which average pool. Using this pooling, we can already create our graph network below. Specifically, we re-use our class `GNNModel` from before, and simply add an average pool and single linear layer for the graph prediction task.

```
[24]: class GraphGNNModel(nn.Module):

    def __init__(self, c_in, c_hidden, c_out, dp_rate_linear=0.5, **kwargs):
        """
        Inputs:
            c_in - Dimension of input features
            c_hidden - Dimension of hidden features
            c_out - Dimension of output features (usually number of classes)
            dp_rate_linear - Dropout rate before the linear layer (usually much higher
                             than inside the GNN)
            kwargs - Additional arguments for the GNNModel object
        """

        super().__init__()

        self.gnn = GNNModel(
            c_in=c_in,
            c_hidden=c_hidden,
            c_out=c_out,
            dp_rate_linear=dp_rate_linear,
            **kwargs
        )
```

(continues on next page)

(continued from previous page)

```

"""
super().__init__()
self.GNN = GNNModel(c_in=c_in,
                     c_hidden=c_hidden,
                     c_out=c_hidden, # Not our prediction output yet!
                     **kwargs)
self.head = nn.Sequential(
    nn.Dropout(dp_rate_linear),
    nn.Linear(c_hidden, c_out)
)

def forward(self, x, edge_index, batch_idx):
    """
    Inputs:
        x - Input features per node
        edge_index - List of vertex index pairs representing the edges in the graph.
        ↪(PyTorch geometric notation)
        batch_idx - Index of batch element for each node
    """
    x = self.GNN(x, edge_index)
    x = geom_nn.global_mean_pool(x, batch_idx) # Average pooling
    x = self.head(x)
    return x

```

Finally, we can create a PyTorch Lightning module to handle the training. It is similar to the modules we have seen before and does nothing surprising in terms of training. As we have a binary classification task, we use the Binary Cross Entropy loss.

```
[25]: class GraphLevelGNN(pl.LightningModule):

    def __init__(self, **model_kwargs):
        super().__init__()
        # Saving hyperparameters
        self.save_hyperparameters()

        self.model = GraphGNNModel(**model_kwargs)
        self.loss_module = nn.BCEWithLogitsLoss() if self.hparams.c_out == 1 else nn.
        ↪CrossEntropyLoss()

    def forward(self, data, mode="train"):
        x, edge_index, batch_idx = data.x, data.edge_index, data.batch
        x = self.model(x, edge_index, batch_idx)
        x = x.squeeze(dim=-1)

        if self.hparams.c_out == 1:
            preds = (x > 0).float()
            data.y = data.y.float()
        else:
            preds = x.argmax(dim=-1)
        loss = self.loss_module(x, data.y)
        acc = (preds == data.y).sum().float() / preds.shape[0]
        return loss, acc
```

(continues on next page)

(continued from previous page)

```

def configure_optimizers(self):
    optimizer = optim.AdamW(self.parameters(), lr=1e-2, weight_decay=0.0) # High lr
    ↵because of small dataset and small model
    return optimizer

def training_step(self, batch, batch_idx):
    loss, acc = self.forward(batch, mode="train")
    self.log('train_loss', loss)
    self.log('train_acc', acc)
    return loss

def validation_step(self, batch, batch_idx):
    _, acc = self.forward(batch, mode="val")
    self.log('val_acc', acc)

def test_step(self, batch, batch_idx):
    _, acc = self.forward(batch, mode="test")
    self.log('test_acc', acc)

```

Below we train the model on our dataset. It resembles the typical training functions we have seen so far.

```
[26]: def train_graph_classifier(model_name, **model_kwargs):
    pl.seed_everything(42)

    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "GraphLevel" + model_name)
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                          callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
                           ↵monitor="val_acc")],
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=500,
                          enable_progress_bar=False)
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't
    ↵need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, f"GraphLevel{model_name}.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = GraphLevelGNN.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything(42)
        model = GraphLevelGNN(c_in=tu_dataset.num_node_features,
                              c_out=1 if tu_dataset.num_classes==2 else tu_dataset.num_
                           ↵classes,
                              **model_kwargs)
    trainer.fit(model, graph_train_loader, graph_val_loader)
    model = GraphLevelGNN.load_from_checkpoint(trainer.checkpoint_callback.best_
                           ↵model_path)

```

(continues on next page)

(continued from previous page)

```
# Test best model on validation and test set
train_result = trainer.test(model, graph_train_loader, verbose=False)
test_result = trainer.test(model, graph_test_loader, verbose=False)
result = {"test": test_result[0]['test_acc'], "train": train_result[0]['test_acc']}
return model, result
```

Finally, let's perform the training and testing. Feel free to experiment with different GNN layers, hyperparameters, etc.

```
[27]: model, result = train_graph_classifier(model_name="GraphConv",
                                             c_hidden=256,
                                             layer_name="GraphConv",
                                             num_layers=3,
                                             dp_rate_linear=0.5,
                                             dp_rate=0.0)

GPU available: True, used: True
I1113 19:12:54.045717 139969460983616 distributed.py:49] GPU available: True, used: True
TPU available: False, using: 0 TPU cores
I1113 19:12:54.047091 139969460983616 distributed.py:49] TPU available: False, using: 0 ↵
TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
I1113 19:12:54.048336 139969460983616 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_
VISIBLE_DEVICES: [0]
/home/phillip/anaconda3/envs/nlp1/lib/python3.7/site-packages/pytorch_lightning/
utilities/distributed.py:45: UserWarning: Your test_dataloader has `shuffle=True`, it ↵
is best practice to turn this off for validation and test dataloaders.
warnings.warn(*args, **kwargs)

Found pretrained model, loading...
```

```
[28]: print(f"Train performance: {100.0*result['train']:4.2f}%")
print(f"Test performance: {100.0*result['test']:4.2f}%")

Train performance: 94.27%
Test performance: 92.11%
```

The test performance shows that we obtain quite good scores on an unseen part of the dataset. It should be noted that as we have been using the test set for validation as well, we might have overfitted slightly to this set. Nevertheless, the experiment shows us that GNNs can be indeed powerful to predict the properties of graphs and/or molecules.

4.10.4 Conclusion

In this tutorial, we have seen the application of neural networks to graph structures. We looked at how a graph can be represented (adjacency matrix or edge list), and discussed the implementation of common graph layers: GCN and GAT. The implementations showed the practical side of the layers, which is often easier than the theory. Finally, we experimented with different tasks, on node-, edge- and graph-level. Overall, we have seen that including graph information in the predictions can be crucial for achieving high performance. There are a lot of applications that benefit from GNNs, and the importance of these networks will likely increase over the next years.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.11 Tutorial 8: Deep Energy-Based Generative Models

Filled notebook:

Pre-trained models:

Recordings:

Author: Phillip Lippe

In this tutorial, we will look at energy-based deep learning models, and focus on their application as generative models. Energy models have been a popular tool before the huge deep learning hype around 2012 hit. However, in recent years, energy-based models have gained increasing attention because of improved training methods and tricks being proposed. Although they are still in a research stage, they have shown to outperform strong Generative Adversarial Networks (Lecture/Tutorial 10) in certain cases which have been the state of the art of generating images ([blog post](#) about strong energy-based models, [blog post](#) about the power of GANs). Hence, it is important to be aware of energy-based models, and as the theory can be abstract sometimes, we will show the idea of energy-based models with a lot of examples.

First, let's import our standard libraries below.

```
[1]: ## Standard libraries
import os
import json
import math
import numpy as np
import random

## Imports for plotting
import matplotlib.pyplot as plt
from matplotlib import cm
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
from mpl_toolkits.mplot3d.axes3d import Axes3D
from mpl_toolkits.mplot3d import proj3d
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
```

(continues on next page)

(continued from previous page)

```

import torch.utils.data as data
import torch.optim as optim
# Torchvision
import torchvision
from torchvision.datasets import MNIST
from torchvision import transforms
# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial8"

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", device)

```

We also have pre-trained models that we download below.

```

[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial8/"
# Files to download
pretrained_files = ["MNIST.ckpt", "tensorboards/events.out.tfevents.MNIST"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)

```

(continues on next page)

(continued from previous page)

```
except HTTPError as e:
    print("Something went wrong. Please try to download the file from the GDrive_
    ↪folder, or contact the author with the full output including the following error:\n",_
    ↪e)
```

4.11.1 Energy Models

In the first part of this tutorial, we will review the theory of the energy-based models (the same theory has been discussed in Lecture 8). While most of the previous models had the goal of classification or regression, energy-based models are motivated from a different perspective: density estimation. Given a dataset with a lot of elements, we want to estimate the probability distribution over the whole data space. As an example, if we model images from CIFAR10, our goal would be to have a probability distribution over all possible images of size $32 \times 32 \times 3$ where those images have a high likelihood that look realistic and are one of the 10 CIFAR classes. Simple methods like interpolation between images don't work because images are extremely high-dimensional (especially for large HD images). Hence, we turn to deep learning methods that have performed well on complex data.

However, how do we predict a probability distribution $p(\mathbf{x})$ over so many dimensions using a simple neural network? The problem is that we cannot just predict a score between 0 and 1, because a probability distribution over data needs to fulfill two properties:

1. The probability distribution needs to assign any possible value of \mathbf{x} a non-negative value: $p(\mathbf{x}) \geq 0$.
2. The probability density must sum/integrate to 1 over **all** possible inputs: $\int_{\mathbf{x}} p(\mathbf{x}) d\mathbf{x} = 1$.

Luckily, there are actually many approaches for this, and one of them are energy-based models. The fundamental idea of energy-based models is that you can turn any function that predicts values larger than zero into a probability distribution by dividing by its volume. Imagine we have a neural network, which has as output a single neuron, like in regression. We can call this network $E_{\theta}(\mathbf{x})$, where θ are our parameters of the network, and \mathbf{x} the input data (e.g. an image). The output of E_{θ} is a scalar value between $-\infty$ and ∞ . Now, we can use basic probability theory to *normalize* the scores of all possible inputs:

$$q_{\theta}(\mathbf{x}) = \frac{\exp(-E_{\theta}(\mathbf{x}))}{Z_{\theta}} \quad \text{where} \quad Z_{\theta} = \begin{cases} \int_{\mathbf{x}} \exp(-E_{\theta}(\mathbf{x})) d\mathbf{x} & \text{if } \mathbf{x} \text{ is continuous} \\ \sum_{\mathbf{x}} \exp(-E_{\theta}(\mathbf{x})) & \text{if } \mathbf{x} \text{ is discrete} \end{cases}$$

The \exp -function ensures that we assign a probability greater than zero to any possible input. We use a negative sign in front of E because we call E_{θ} to be the energy function: data points with high likelihood have a low energy, while data points with low likelihood have a high energy. Z_{θ} is our normalization terms that ensures that the density integrates/sums to 1. We can show this by integrating over $q_{\theta}(\mathbf{x})$:

$$\int_{\mathbf{x}} q_{\theta}(\mathbf{x}) d\mathbf{x} = \int_{\mathbf{x}} \frac{\exp(-E_{\theta}(\mathbf{x}))}{\int_{\tilde{\mathbf{x}}} \exp(-E_{\theta}(\tilde{\mathbf{x}})) d\tilde{\mathbf{x}}} d\mathbf{x} = \frac{\int_{\mathbf{x}} \exp(-E_{\theta}(\mathbf{x})) d\mathbf{x}}{\int_{\tilde{\mathbf{x}}} \exp(-E_{\theta}(\tilde{\mathbf{x}})) d\tilde{\mathbf{x}}} = 1$$

Note that we call the probability distribution $q_{\theta}(\mathbf{x})$ because this is the learned distribution by the model, and is trained to be as close as possible to the *true*, unknown distribution $p(\mathbf{x})$.

The main benefit of this formulation of the probability distribution is its great flexibility as we can choose E_{θ} in whatever way we like, without any constraints. Nevertheless, when looking at the equation above, we can see a fundamental issue: How do we calculate Z_{θ} ? There is no chance that we can calculate Z_{θ} analytically for high-dimensional input and/or larger neural networks, but the task requires us to know Z_{θ} . Although we can't determine the exact likelihood of a point, there exist methods with which we can train energy-based models. Thus, we will look next at "Contrastive Divergence" for training the model.

Contrastive Divergence

When we train a model on generative modeling, it is usually done by maximum likelihood estimation. In other words, we try to maximize the likelihood of the examples in the training set. As the exact likelihood of a point cannot be determined due to the unknown normalization constant Z_θ , we need to train energy-based models slightly different. We cannot just maximize the un-normalized probability $\exp(-E_\theta(\mathbf{x}_{\text{train}}))$ because there is no guarantee that Z_θ stays constant, or that $\mathbf{x}_{\text{train}}$ is becoming more likely than the others. However, if we base our training on comparing the likelihood of points, we can create a stable objective. Namely, we can re-write our maximum likelihood objective where we maximize the probability of $\mathbf{x}_{\text{train}}$ compared to a randomly sampled data point of our model:

$$\begin{aligned}\nabla_\theta \mathcal{L}_{\text{MLE}}(\theta; p) &= -\mathbb{E}_{p(\mathbf{x})} [\nabla_\theta \log q_\theta(\mathbf{x})] \\ &= \mathbb{E}_{p(\mathbf{x})} [\nabla_\theta E_\theta(\mathbf{x})] - \mathbb{E}_{q_\theta(\mathbf{x})} [\nabla_\theta E_\theta(\mathbf{x})]\end{aligned}$$

Note that the loss is still an objective we want to minimize. Thus, we try to minimize the energy for data points from the dataset, while maximizing the energy for randomly sampled data points from our model (how we sample will be explained below). Although this objective sounds intuitive, how is it actually derived from our original distribution $q_\theta(\mathbf{x})$? The trick is that we approximate Z_θ by a single Monte-Carlo sample. This gives us the exact same objective as written above.

Visually, we can look at the objective as follows (figure credit - [Stefano Ermon and Aditya Grover](#)):

f_θ represents $\exp(-E_\theta(\mathbf{x}))$ in our case. The point on the right, called “correct answer”, represents a data point from the dataset (i.e. x_{train}), and the left point, “wrong answer”, a sample from our model (i.e. x_{sample}). Thus, we try to “pull up” the probability of the data points in the dataset, while “pushing down” randomly sampled points. The two forces for pulling and pushing are in balance iff $q_\theta(\mathbf{x}) = p(\mathbf{x})$.

Sampling from Energy-Based Models

For sampling from an energy-based model, we can apply a Markov Chain Monte Carlo using Langevin Dynamics. The idea of the algorithm is to start from a random point, and slowly move towards the direction of higher probability using the gradients of E_θ . Nevertheless, this is not enough to fully capture the probability distribution. We need to add noise ω at each gradient step to the current sample. Under certain conditions such as that we perform the gradient steps an infinite amount of times, we would be able to create an exact sample from our modeled distribution. However, as this is not practically possible, we usually limit the chain to K steps (K a hyperparameter that needs to be finetuned). Overall, the sampling procedure can be summarized in the following algorithm:

Applications of Energy-based models beyond generation

Modeling the probability distribution for sampling new data is not the only application of energy-based models. Any application which requires us to compare two elements is much simpler to learn because we just need to go for the higher energy. A couple of examples are shown below (figure credit - [Stefano Ermon and Aditya Grover](#)). A classification setup like object recognition or sequence labeling can be considered as an energy-based task as we just need to find the Y input that minimizes the output $E(X, Y)$ (hence maximizes probability). Similarly, a popular application of energy-based models is denoising of images. Given an image X with a lot of noise, we try to minimize the energy by finding the true input image Y .

Nonetheless, we will focus on generative modeling here as in the next couple of lectures, we will discuss more generative deep learning approaches.

4.11.2 Image generation

As an example for energy-based models, we will train a model on image generation. Specifically, we will look at how we can generate MNIST digits with a very simple CNN model. However, it should be noted that energy models are not easy to train and often diverge if the hyperparameters are not well tuned. We will rely on training tricks proposed in the paper [Implicit Generation and Generalization in Energy-Based Models](#) by Yilun Du and Igor Mordatch ([blog](#)). The important part of this notebook is however to see how the theory above can actually be used in a model.

Dataset

First, we can load the MNIST dataset below. Note that we need to normalize the images between -1 and 1 instead of mean 0 and std 1 because during sampling, we have to limit the input space. Scaling between -1 and 1 makes it easier to implement it.

```
[3]: # Transformations applied on each image => make them a tensor and normalize between -1 and 1
transform = transforms.Compose([transforms.ToTensor(),
                               transforms.Normalize((0.5,), (0.5,))])

# Loading the training dataset. We need to split it into a training and validation part
train_set = MNIST(root=DATASET_PATH, train=True, transform=transform, download=True)

# Loading the test set
test_set = MNIST(root=DATASET_PATH, train=False, transform=transform, download=True)

# We define a set of data loaders that we can use for various purposes later.
# Note that for actually training a model, we will use different data loaders
# with a lower batch size.
train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True,
                                num_workers=4, pin_memory=True)
test_loader = data.DataLoader(test_set, batch_size=256, shuffle=False, drop_last=False,
                                num_workers=4)
```

CNN Model

First, we implement our CNN model. The MNIST images are of size 28x28, hence we only need a small model. As an example, we will apply several convolutions with stride 2 that downscale the images. If you are interested, you can also use a deeper model such as a small ResNet, but for simplicity, we will stick with the tiny network.

It is a good practice to use a smooth activation function like Swish instead of ReLU in the energy model. This is because we will rely on the gradients we get back with respect to the input image, which should not be sparse.

```
[4]: class Swish(nn.Module):

    def forward(self, x):
        return x * torch.sigmoid(x)

class CNNModel(nn.Module):

    def __init__(self, hidden_features=32, out_dim=1, **kwargs):
```

(continues on next page)

(continued from previous page)

```

super().__init__()
# We increase the hidden dimension over layers. Here pre-calculated for
# simplicity.
c_hid1 = hidden_features//2
c_hid2 = hidden_features
c_hid3 = hidden_features*2

# Series of convolutions and Swish activation functions
self.cnn_layers = nn.Sequential(
    nn.Conv2d(1, c_hid1, kernel_size=5, stride=2, padding=4), # [16x16] -_
    # Larger padding to get 32x32 image
    Swish(),
    nn.Conv2d(c_hid1, c_hid2, kernel_size=3, stride=2, padding=1), # [8x8]
    Swish(),
    nn.Conv2d(c_hid2, c_hid3, kernel_size=3, stride=2, padding=1), # [4x4]
    Swish(),
    nn.Conv2d(c_hid3, c_hid3, kernel_size=3, stride=2, padding=1), # [2x2]
    Swish(),
    nn.Flatten(),
    nn.Linear(c_hid3*4, c_hid3),
    Swish(),
    nn.Linear(c_hid3, out_dim)
)

def forward(self, x):
    x = self.cnn_layers(x).squeeze(dim=-1)
    return x

```

In the rest of the notebook, the output of the model will actually not represent $E_\theta(\mathbf{x})$, but $-E_\theta(\mathbf{x})$. This is a standard implementation practice for energy-based models, as some people also write the energy probability density as $q_\theta(\mathbf{x}) = \frac{\exp(f_\theta(\mathbf{x}))}{Z_\theta}$. In that case, the model would actually represent $f_\theta(\mathbf{x})$. In the training loss etc., we need to be careful to not switch up the signs.

Sampling buffer

In the next part, we look at the training with sampled elements. To use the contrastive divergence objective, we need to generate samples during training. Previous work has shown that due to the high dimensionality of images, we need a lot of iterations inside the MCMC sampling to obtain reasonable samples. However, there is a training trick that significantly reduces the sampling cost: using a sampling buffer. The idea is that we store the samples of the last couple of batches in a buffer, and re-use those as the starting point of the MCMC algorithm for the next batches. This reduces the sampling cost because the model requires a significantly lower number of steps to converge to reasonable samples. However, to not solely rely on previous samples and allow novel samples as well, we re-initialize 5% of our samples from scratch (random noise between -1 and 1).

Below, we implement the sampling buffer. The function `sample_new_exmps` returns a new batch of “fake” images. We refer to those as fake images because they have been generated, but are not actually part of the dataset. As mentioned before, we use initialize 5% randomly, and 95% are randomly picked from our buffer. On this initial batch, we perform MCMC for 60 iterations to improve the image quality and come closer to samples from $q_\theta(\mathbf{x})$. In the function `generate_samples`, we implemented the MCMC for images. Note that the hyperparameters of `step_size`, `steps`, the noise standard deviation σ are specifically set for MNIST, and need to be finetuned for a different dataset if you want to use such.

```
[5]: class Sampler:

    def __init__(self, model, img_shape, sample_size, max_len=8192):
        """
        Inputs:
            model - Neural network to use for modeling E_theta
            img_shape - Shape of the images to model
            sample_size - Batch size of the samples
            max_len - Maximum number of data points to keep in the buffer
        """
        super().__init__()
        self.model = model
        self.img_shape = img_shape
        self.sample_size = sample_size
        self.max_len = max_len
        self.examples = [(torch.rand((1,) + img_shape) * 2 - 1) for _ in range(self.sample_size)]

    def sample_new_exmps(self, steps=60, step_size=10):
        """
        Function for getting a new batch of "fake" images.
        Inputs:
            steps - Number of iterations in the MCMC algorithm
            step_size - Learning rate nu in the algorithm above
        """
        # Choose 95% of the batch from the buffer, 5% generate from scratch
        n_new = np.random.binomial(self.sample_size, 0.05)
        rand_imgs = torch.rand((n_new,) + self.img_shape) * 2 - 1
        old_imgs = torch.cat(random.choices(self.examples, k=self.sample_size - n_new), dim=0)
        inp_imgs = torch.cat([rand_imgs, old_imgs], dim=0).detach().to(device)

        # Perform MCMC sampling
        inp_imgs = Sampler.generate_samples(self.model, inp_imgs, steps=steps, step_size=step_size)

        # Add new images to the buffer and remove old ones if needed
        self.examples = list(inp_imgs.to(torch.device("cpu")).chunk(self.sample_size, dim=0)) + self.examples
        self.examples = self.examples[:self.max_len]
        return inp_imgs

    @staticmethod
    def generate_samples(model, inp_imgs, steps=60, step_size=10, return_img_per_step=False):
        """
        Function for sampling images for a given model.
        Inputs:
            model - Neural network to use for modeling E_theta
            inp_imgs - Images to start from for sampling. If you want to generate new images, enter noise between -1 and 1.
            steps - Number of iterations in the MCMC algorithm.
            step_size - Learning rate nu in the algorithm above
        """

```

(continues on next page)

(continued from previous page)

```

    return_img_per_step - If True, we return the sample at every iteration of
→the MCMC
"""
# Before MCMC: set model parameters to "required_grad=False"
# because we are only interested in the gradients of the input.
is_training = model.training
model.eval()
for p in model.parameters():
    p.requires_grad = False
inp_imgs.requires_grad = True

# Enable gradient calculation if not already the case
had_gradients_enabled = torch.is_grad_enabled()
torch.set_grad_enabled(True)

# We use a buffer tensor in which we generate noise each loop iteration.
# More efficient than creating a new tensor every iteration.
noise = torch.randn(inp_imgs.shape, device=inp_imgs.device)

# List for storing generations at each step (for later analysis)
imgs_per_step = []

# Loop over K (steps)
for _ in range(steps):
    # Part 1: Add noise to the input.
    noise.normal_(0, 0.005)
    inp_imgs.data.add_(noise.data)
    inp_imgs.data.clamp_(min=-1.0, max=1.0)

    # Part 2: calculate gradients for the current input.
    out_imgs = -model(inp_imgs)
    out_imgs.sum().backward()
    inp_imgs.grad.data.clamp_(-0.03, 0.03) # For stabilizing and preventing too
→high gradients

    # Apply gradients to our current samples
    inp_imgs.data.add_(-step_size * inp_imgs.grad.data)
    inp_imgs.grad.detach_()
    inp_imgs.grad.zero_()
    inp_imgs.data.clamp_(min=-1.0, max=1.0)

    if return_img_per_step:
        imgs_per_step.append(inp_imgs.clone().detach())

# Reactivate gradients for parameters for training
for p in model.parameters():
    p.requires_grad = True
model.train(is_training)

# Reset gradient calculation to setting before this function
torch.set_grad_enabled(had_gradients_enabled)

```

(continues on next page)

(continued from previous page)

```

if return_img_per_step:
    return torch.stack(imgs_per_step, dim=0)
else:
    return inp_imgs

```

The idea of the buffer becomes a bit clearer in the following algorithm.

Training algorithm

With the sampling buffer being ready, we can complete our training algorithm. Below is shown a summary of the full training algorithm of an energy model on image modeling:

The first few statements in each training iteration concern the sampling of the real and fake data, as we have seen above with the sample buffer. Next, we calculate the contrastive divergence objective using our energy model E_θ . However, one additional training trick we need is to add a regularization loss on the output of E_θ . As the output of the network is not constrained and adding a large bias or not to the output doesn't change the contrastive divergence loss, we need to ensure somehow else that the output values are in a reasonable range. Without the regularization loss, the output values will fluctuate in a very large range. With this, we ensure that the values for the real data are around 0, and the fake data likely slightly lower (for noise or outliers the score can be still significantly lower). As the regularization loss is less important than the Contrastive Divergence, we have a weight factor α which is usually quite some smaller than 1. Finally, we perform an update step with an optimizer on the combined loss and add the new samples to the buffer.

Below, we put this training dynamic into a PyTorch Lightning module. Remember that, since we model $f_\theta(x) = -E_\theta(x)$, we need to be careful with switching all important signs, e.g. in the loss function.

```

[6]: class DeepEnergyModel(pl.LightningModule):

    def __init__(self, img_shape, batch_size, alpha=0.1, lr=1e-4, beta1=0.0, **CNN_args):
        super().__init__()
        self.save_hyperparameters()

        self.cnn = CNNModel(**CNN_args)
        self.sampler = Sampler(self.cnn, img_shape=img_shape, sample_size=batch_size)
        self.example_input_array = torch.zeros(1, *img_shape)

    def forward(self, x):
        z = self.cnn(x)
        return z

    def configure_optimizers(self):
        # Energy models can have issues with momentum as the loss surfaces changes with its parameters.
        # Hence, we set it to 0 by default.
        optimizer = optim.Adam(self.parameters(), lr=self.hparams.lr, betas=(self.hparams.beta1, 0.999))
        scheduler = optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.97) # Exponential decay over epochs
        return [optimizer], [scheduler]

    def training_step(self, batch, batch_idx):
        # We add minimal noise to the original images to prevent the model from focusing on purely "clean" inputs

```

(continues on next page)

(continued from previous page)

```

real_imgs, _ = batch
small_noise = torch.randn_like(real_imgs) * 0.005
real_imgs.add_(small_noise).clamp_(min=-1.0, max=1.0)

# Obtain samples
fake_imgs = self.sampler.sample_new_exmps(steps=60, step_size=10)

# Predict energy score for all images
inp_imgs = torch.cat([real_imgs, fake_imgs], dim=0)
real_out, fake_out = self.cnn(inp_imgs).chunk(2, dim=0)

# Calculate losses
reg_loss = self.hparams.alpha * (real_out ** 2 + fake_out ** 2).mean()
cdiv_loss = fake_out.mean() - real_out.mean()
loss = reg_loss + cdiv_loss

# Logging
self.log('loss', loss)
self.log('loss_regularization', reg_loss)
self.log('loss_contrastive_divergence', cdiv_loss)
self.log('metrics_avg_real', real_out.mean())
self.log('metrics_avg_fake', fake_out.mean())
return loss

def validation_step(self, batch, batch_idx):
    # For validating, we calculate the contrastive divergence between purely random
    # images and unseen examples
    # Note that the validation/test step of energy-based models depends on what we
    # are interested in the model
    real_imgs, _ = batch
    fake_imgs = torch.rand_like(real_imgs) * 2 - 1

    inp_imgs = torch.cat([real_imgs, fake_imgs], dim=0)
    real_out, fake_out = self.cnn(inp_imgs).chunk(2, dim=0)

    cdiv = fake_out.mean() - real_out.mean()
    self.log('val_contrastive_divergence', cdiv)
    self.log('val_fake_out', fake_out.mean())
    self.log('val_real_out', real_out.mean())

```

We do not implement a test step because energy-based, generative models are usually not evaluated on a test set. The validation step however is used to get an idea of the difference between ennergy/lielihood of random images to unseen examples of the dataset. Alternative test steps would be to generate new images and evaluate how realistic they are based on FID or Inception score, or try to denoise images.

Callbacks

To track the performance of our model during training, we will make extensive use of PyTorch Lightning's callback framework. Remember that callbacks can be used for running small functions at any point of the training, for instance after finishing an epoch. Here, we will use three different callbacks we define ourselves.

The first callback, called `GenerateCallback`, is used for adding image generations to the model during training. After every N epochs (usually $N = 5$ to reduce output to TensorBoard), we take a small batch of random images and perform many MCMC iterations until the model's generation converges. Compared to the training that used 60 iterations, we use 256 here because (1) we only have to do it once compared to the training that has to do it every iteration, and (2) we do not start from a buffer here, but from scratch. It is implemented as follows:

```
[7]: class GenerateCallback(pl.Callback):

    def __init__(self, batch_size=8, vis_steps=8, num_steps=256, every_n_epochs=5):
        super().__init__()
        self.batch_size = batch_size           # Number of images to generate
        self.vis_steps = vis_steps            # Number of steps within generation to_
→ visualize
        self.num_steps = num_steps           # Number of steps to take during generation
        self.every_n_epochs = every_n_epochs # Only save those images every N epochs_
→ (otherwise tensorboard gets quite large)

    def on_epoch_end(self, trainer, pl_module):
        # Skip for all other epochs
        if trainer.current_epoch % self.every_n_epochs == 0:
            # Generate images
            imgs_per_step = self.generate_imgs(pl_module)
            # Plot and add to tensorboard
            for i in range(imgs_per_step.shape[1]):
                step_size = self.num_steps // self.vis_steps
                imgs_to_plot = imgs_per_step[step_size-1::step_size,i]
                grid = torchvision.utils.make_grid(imgs_to_plot, nrow=imgs_to_plot.
→ shape[0], normalize=True, range=(-1,1))
                    trainer.logger.experiment.add_image(f"generation_{i}", grid, global_
→ step=trainer.current_epoch)

    def generate_imgs(self, pl_module):
        pl_module.eval()
        start_imgs = torch.rand((self.batch_size,) + pl_module.hparams["img_shape"]).
→ to(pl_module.device)
        start_imgs = start_imgs * 2 - 1
        torch.set_grad_enabled(True) # Tracking gradients for sampling necessary
        imgs_per_step = Sampler.generate_samples(pl_module.cnn, start_imgs, steps=self.
→ num_steps, step_size=10, return_img_per_step=True)
        torch.set_grad_enabled(False)
        pl_module.train()
        return imgs_per_step
```

The second callback is called `SamplerCallback`, and simply adds a randomly picked subset of images in the sampling buffer to the TensorBoard. This helps to understand what images are currently shown to the model as “fake”.

```
[8]: class SamplerCallback(pl.Callback):
```

(continues on next page)

(continued from previous page)

```

def __init__(self, num_imgs=32, every_n_epochs=5):
    super().__init__()
    self.num_imgs = num_imgs           # Number of images to plot
    self.every_n_epochs = every_n_epochs # Only save those images every N epochs
    →(otherwise tensorboard gets quite large)

    def on_epoch_end(self, trainer, pl_module):
        if trainer.current_epoch % self.every_n_epochs == 0:
            exmp_imgs = torch.cat(random.choices(pl_module.sampler.examples, k=self.num_
            →imgs), dim=0)
            grid = torchvision.utils.make_grid(exmp_imgs, nrow=4, normalize=True, ←
            →range=(-1,1))
            trainer.logger.experiment.add_image("sampler", grid, global_step=trainer.
            →current_epoch)

```

Finally, our last callback is `OutlierCallback`. This callback evaluates the model by recording the (negative) energy assigned to random noise. While our training loss is almost constant across iterations, this score is likely showing the progress of the model to detect “outliers”.

```
[9]: class OutlierCallback(pl.Callback):

    def __init__(self, batch_size=1024):
        super().__init__()
        self.batch_size = batch_size

    def on_epoch_end(self, trainer, pl_module):
        with torch.no_grad():
            pl_module.eval()
            rand_imgs = torch.rand((self.batch_size,) + pl_module.hparams["img_shape"]).
            →to(pl_module.device)
            rand_imgs = rand_imgs * 2 - 1.0
            rand_out = pl_module.cnn(rand_imgs).mean()
            pl_module.train()

            trainer.logger.experiment.add_scalar("rand_out", rand_out, global_step=trainer.
            →current_epoch)
```

Running the model

Finally, we can add everything together to create our final training function. The function is very similar to any other PyTorch Lightning training function we have seen so far. However, there is the small difference of that we do not test the model on a test set because we will analyse the model afterward by checking its prediction and ability to perform outlier detection.

```
[10]: def train_model(**kwargs):
    # Create a PyTorch Lightning trainer with the generation callback
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "MNIST"),
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=60,
                          gradient_clip_val=0.1,
```

(continues on next page)

(continued from previous page)

```

        callbacks=[ModelCheckpoint(save_weights_only=True, mode="min",
        monitor='val_contrastive_divergence'),
                    GenerateCallback(every_n_epochs=5),
                    SamplerCallback(every_n_epochs=5),
                    OutlierCallback(),
                    LearningRateMonitor("epoch")
                ])
# Check whether pretrained model exists. If yes, load it and skip training
pretrained_filename = os.path.join(CHECKPOINT_PATH, "MNIST.ckpt")
if os.path.isfile(pretrained_filename):
    print("Found pretrained model, loading...")
    model = DeepEnergyModel.load_from_checkpoint(pretrained_filename)
else:
    pl.seed_everything(42)
    model = DeepEnergyModel(**kwargs)
    trainer.fit(model, train_loader, test_loader)
    model = DeepEnergyModel.load_from_checkpoint(trainer.checkpoint_callback.best_
model_path)
    # No testing as we are more interested in other properties
return model

```

```
[11]: model = train_model(img_shape=(1, 28, 28),
                        batch_size=train_loader.batch_size,
                        lr=1e-4,
                        beta1=0.0)

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Found pretrained model, loading...
```

4.11.3 Analysis

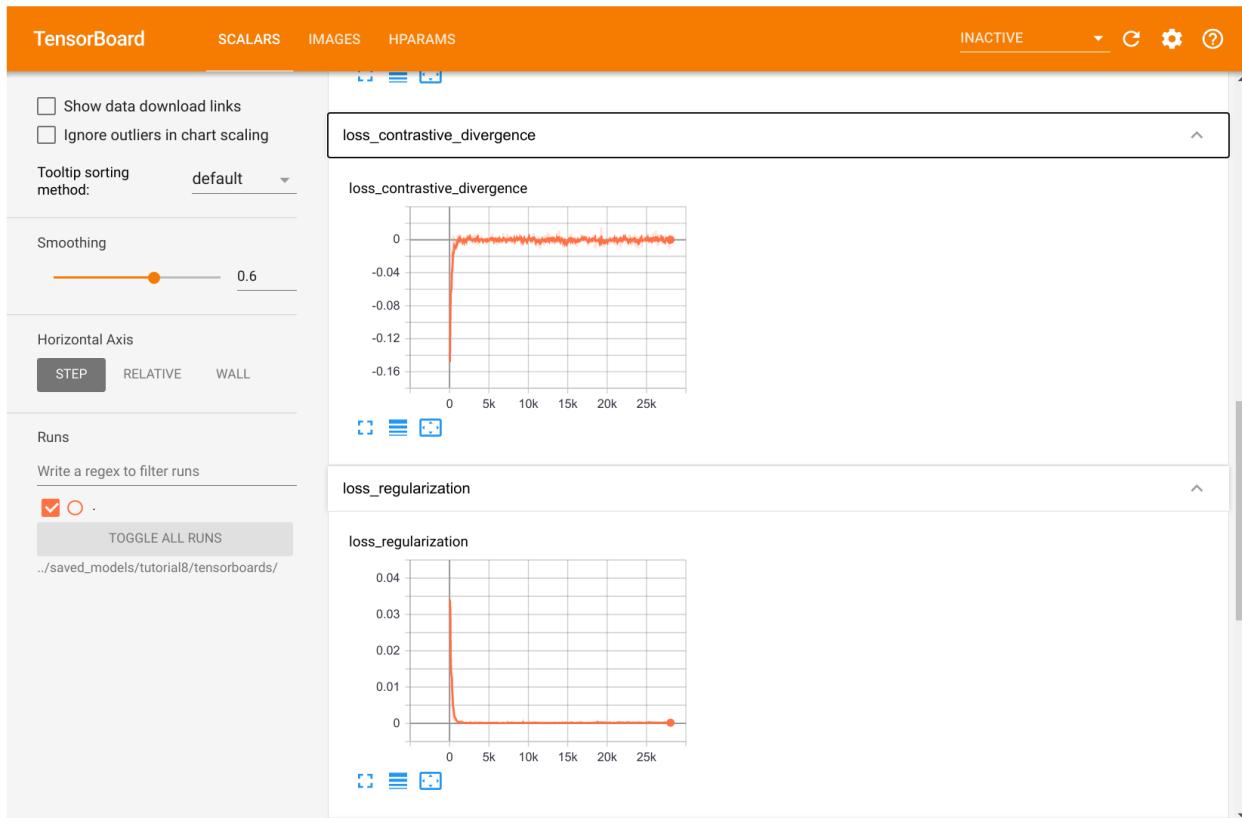
In the last part of the notebook, we will try to take the trained energy-based generative model, and analyse its properties.

TensorBoard

The first thing we can look at is the TensorBoard generate during training. This can help us to understand the training dynamic even better, and shows potential issues. Let's load the TensorBoard below:

```
[12]: # Import tensorboard
%load_ext tensorboard

[13]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH!
%tensorboard --logdir ../saved_models/tutorial8/tensorboards/
```



We see that the contrastive divergence as well as the regularization converge quickly to 0. However, the training continues although the loss is always close to zero. This is because our “training” data changes with the model by sampling. The progress of training can be best measured by looking at the samples across iterations, and the score for random images that decreases constantly over time.

Image Generation

Another way of evaluating generative models is by sampling a few generated images. Generative models need to be good at generating realistic images as this truly shows that they have modeled the true data distribution. Thus, let’s sample a few images of the model below:

```
[14]: model.to(device)
pl.seed_everything(43)
callback = GenerateCallback(batch_size=4, vis_steps=8, num_steps=256)
imgs_per_step = callback.generate_imgs(model)
imgs_per_step = imgs_per_step.cpu()
```

The characteristic of sampling with energy-based models is that they require the iterative MCMC algorithm. To gain an insight in how the images change over iterations, we plot a few intermediate samples in the MCMC as well:

```
[15]: for i in range(imgs_per_step.shape[1]):
    step_size = callback.num_steps // callback.vis_steps
    imgs_to_plot = imgs_per_step[step_size-1::step_size,i]
    imgs_to_plot = torch.cat([imgs_per_step[0:1,i], imgs_to_plot], dim=0)
    grid = torchvision.utils.make_grid(imgs_to_plot, nrow=imgs_to_plot.shape[0],_
    normalize=True, range=(-1,1), pad_value=0.5, padding=2)
```

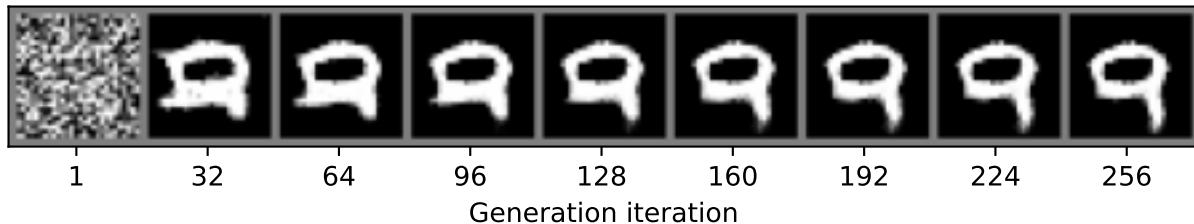
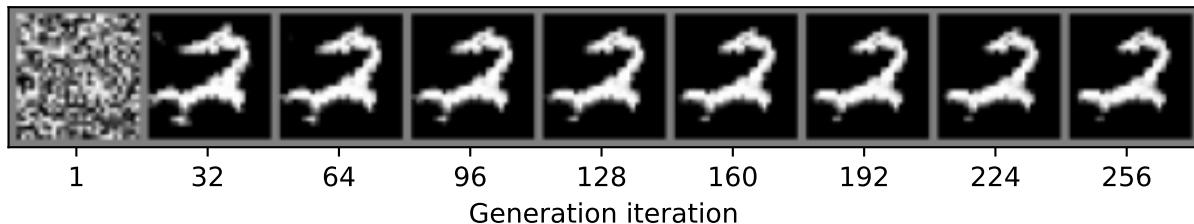
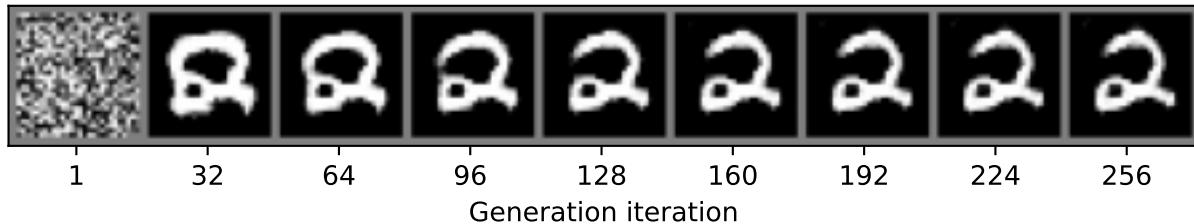
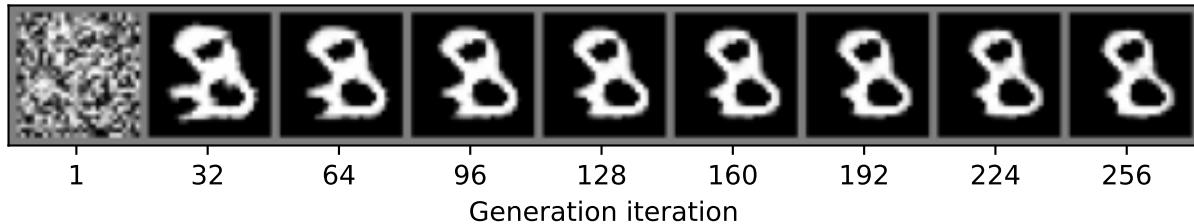
(continues on next page)

(continued from previous page)

```

grid = grid.permute(1, 2, 0)
plt.figure(figsize=(8,8))
plt.imshow(grid)
plt.xlabel("Generation iteration")
plt.xticks([(imgs_per_step.shape[-1]+2)*(0.5+j) for j in range(callback.vis_steps+1)], labels=[1] + list(range(step_size, imgs_per_step.shape[0]+1, step_size)))
plt.yticks([])
plt.show()

```



We see that although starting from noise in the very first step, the sampling algorithm obtains reasonable shapes after only 32 steps. Over the next 200 steps, the shapes become clearer and changed towards realistic digits. The specific samples can differ when you run the code on Colab, hence the following description is specific to the plots shown on the website. The first row shows an 8, where we remove unnecessary white parts over iterations. The transformation across iterations can be seen at best for the second sample, which creates a digit of 2. While the first sample after 32 iterations looks a bit like a digit, but not really, the sample is transformed more and more to a typical image of the digit 2.

Out-of-distribution detection

A very common and strong application of energy-based models is out-of-distribution detection (sometimes referred to as “anomaly” detection). As more and more deep learning models are applied in production and applications, a crucial aspect of these models is to know what the models don’t know. Deep learning models are usually overconfident, meaning that they classify even random images sometimes with 100% probability. Clearly, this is not something that we want to see in applications. Energy-based models can help with this problem because they are trained to detect images that do not fit the training dataset distribution. Thus, in those applications, you could train an energy-based model along with the classifier, and only output predictions if the energy-based models assign a (unnormalized) probability higher than δ to the image. You can actually combine classifiers and energy-based objectives in a single model, as proposed in this [paper](#).

In this part of the analysis, we want to test the out-of-distribution capability of our energy-based model. Remember that a lower output of the model denotes a low probability. Thus, we hope to see low scores if we enter random noise to the model:

```
[16]: with torch.no_grad():
    rand_imgs = torch.rand((128,) + model.hparams.img_shape).to(model.device)
    rand_imgs = rand_imgs * 2 - 1.0
    rand_out = model.cnn(rand_imgs).mean()
    print(f"Average score for random images: {rand_out.item():.2f}")

Average score for random images: -17.88
```

As we hoped, the model assigns very low probability to those noisy images. As another reference, let’s look at predictions for a batch of images from the training set:

```
[17]: with torch.no_grad():
    train_imgs, _ = next(iter(train_loader))
    train_imgs = train_imgs.to(model.device)
    train_out = model.cnn(train_imgs).mean()
    print(f"Average score for training images: {train_out.item():.2f}")

Average score for training images: -0.00
```

The scores are close to 0 because of the regularization objective that was added to the training. So clearly, the model can distinguish between noise and real digits. However, what happens if we change the training images a little, and see which ones gets a very low score?

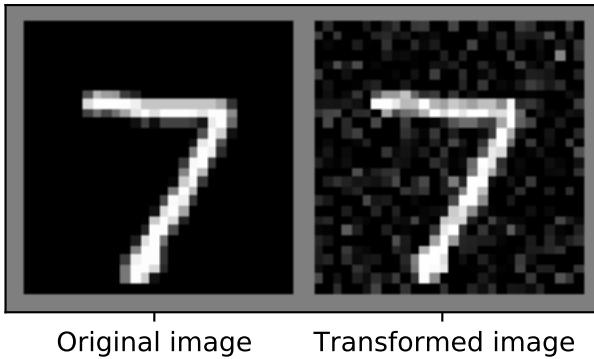
```
[18]: @torch.no_grad()
def compare_images(img1, img2):
    imgs = torch.stack([img1, img2], dim=0).to(model.device)
    score1, score2 = model.cnn(imgs).cpu().chunk(2, dim=0)
    grid = torchvision.utils.make_grid([img1.cpu(), img2.cpu()], nrow=2, normalize=True, range=(-1,1), pad_value=0.5, padding=2)
    grid = grid.permute(1, 2, 0)
    plt.figure(figsize=(4,4))
    plt.imshow(grid)
    plt.xticks([(img1.shape[2]+2)*(0.5+j) for j in range(2)], labels=["Original image", "Transformed image"])
    plt.yticks([])
    plt.show()
    print(f"Score original image: {score1:.2f}")
    print(f"Score transformed image: {score2:.2f}")
```

We use a random test image for this. Feel free to change it to experiment with the model yourself.

```
[19]: test_imgs, _ = next(iter(test_loader))
exmp_img = test_imgs[0].to(model.device)
```

The first transformation is to add some random noise to the image:

```
[20]: img_noisy = exmp_img + torch.randn_like(exmp_img) * 0.3
img_noisy.clamp_(min=-1.0, max=1.0)
compare_images(exmp_img, img_noisy)
```



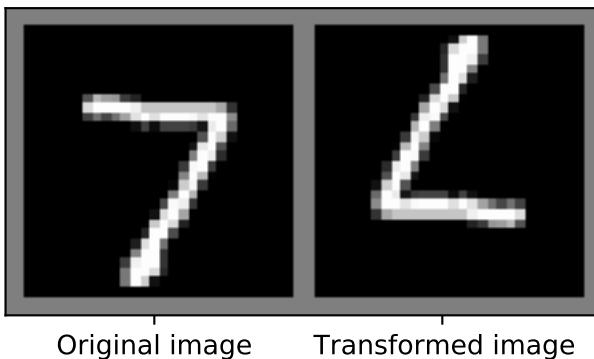
Original image Transformed image

```
Score original image: 0.03
Score transformed image: -0.07
```

We can see that the score considerably drops. Hence, the model can detect random Gaussian noise on the image. This is also to expect as initially, the “fake” samples are pure noise images.

Next, we flip an image and check how this influences the score:

```
[21]: img_flipped = exmp_img.flip(dims=(1,2))
compare_images(exmp_img, img_flipped)
```



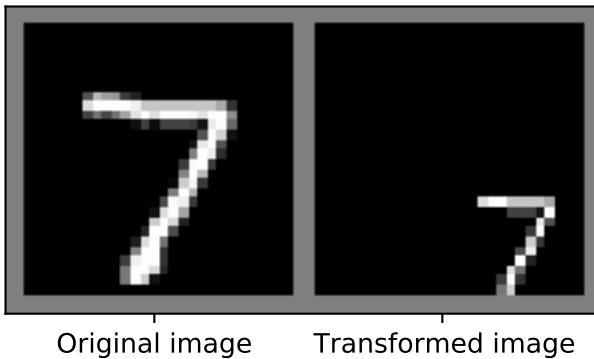
Original image Transformed image

```
Score original image: 0.03
Score transformed image: -0.00
```

If the digit can only be read in this way, for example, the 7, then we can see that the score drops. However, the score only drops slightly. This is likely because of the small size of our model. Keep in mind that generative modeling is a much harder task than classification, as we do not only need to distinguish between classes but learn **all** details/characteristics of the digits. With a deeper model, this could eventually be captured better (but at the cost of greater training instability).

Finally, we check what happens if we reduce the digit significantly in size:

```
[22]: img_tiny = torch.zeros_like(exmp_img) - 1
img_tiny[:,exmp_img.shape[1]//2:,exmp_img.shape[2]//2:] = exmp_img[:,::2,::2]
compare_images(exmp_img, img_tiny)
```



```
Score original image: 0.03
Score transformed image: -0.02
```

The score again drops but not by a large margin, although digits in the MNIST dataset usually are much larger.

Overall, we can conclude that our model is good for detecting Gaussian noise and smaller transformations to existing digits. Nonetheless, to obtain a very good out-of-distribution model, we would need to train deeper models and for more iterations.

Instability

Finally, we should discuss the possible instabilities of energy-based models, in particular for the example of image generation that we have implemented in this notebook. In the process of hyperparameter search for this notebook, there have been several models that diverged. Divergence in energy-based models means that the models assign a high probability to examples of the training set which is a good thing. However, at the same time, the sampling algorithm fails and only generates noise images that obtain minimal probability scores. This happens because the model has created many local maxima in which the generated noise images fall. The energy surface over which we calculate the gradients to reach data points with high probability has “diverged” and is not useful for our MCMC sampling.

Besides finding the optimal hyperparameters, a common trick in energy-based models is to reload stable checkpoints. If we detect that the model is diverging, we stop the training, load the model from one epoch ago where it did not diverge yet. Afterward, we continue training and hope that with a different seed the model is not diverging again. Nevertheless, this should be considered as the “last hope” for stabilizing the models, and careful hyperparameter tuning is the better way to do so. Sensitive hyperparameters include `step_size`, `steps` and the noise standard deviation in the sampler, and the learning rate and feature dimensionality in the CNN model.

4.11.4 Conclusion

In this tutorial, we have discussed energy-based models for generative modeling. The concept relies on the idea that any strictly positive function can be turned into a probability distribution by normalizing over the whole dataset. As this is not reasonable to calculate for high dimensional data like images, we train the model using contrastive divergence and sampling via MCMC. While the idea allows us to turn any neural network into an energy-based model, we have seen that there are multiple training tricks needed to stabilize the training. Furthermore, the training time of these models is relatively long as, during every training iteration, we need to sample new “fake” images, even with a sampling buffer. In the next lectures and assignment, we will see different generative models (e.g. VAE, GAN, NF) that allow us to do generative modeling more stably, but with the cost of more parameters.

If you found this tutorial helpful, consider -ing our repository.
 For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.12 Tutorial 9: Deep Autoencoders

Filled notebook:

Pre-trained models:

Recordings:

JAX+Flax version:

Author: Phillip Lippe

Note: Interested in JAX? Check out our [JAX+Flax version](#) of this tutorial!

In this tutorial, we will take a closer look at autoencoders (AE). Autoencoders are trained on encoding input data such as images into a smaller feature vector, and afterward, reconstruct it by a second neural network, called a decoder. The feature vector is called the “bottleneck” of the network as we aim to compress the input data into a smaller amount of features. This property is useful in many applications, in particular in compressing data or comparing images on a metric beyond pixel-level comparisons. Besides learning about the autoencoder framework, we will also see the “deconvolution” (or transposed convolution) operator in action for scaling up feature maps in height and width. Such deconvolution networks are necessary wherever we start from a small feature vector and need to output an image of full size (e.g. in VAE, GANs, or super-resolution applications).

First of all, we again import most of our standard libraries. We will use [PyTorch Lightning](#) to reduce the training code overhead.

```
[1]: ## Standard libraries
import os
import json
import math
import numpy as np

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
```

(continues on next page)

(continued from previous page)

```

import seaborn as sns
sns.reset_orig()
sns.set()

## Progress bar
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
# Torchvision
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms
# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Tensorboard extension (for visualization purposes later)
from torch.utils.tensorboard import SummaryWriter
%load_ext tensorboard

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial9"

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", device)

```

We have 4 pretrained models that we have to download. Remember to adjust the variables DATASET_PATH and CHECKPOINT_PATH if needed.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial9/"
```

(continues on next page)

(continued from previous page)

```
# Files to download
pretrained_files = ["cifar10_64.ckpt", "cifar10_128.ckpt", "cifar10_256.ckpt", "cifar10_
↪384.ckpt"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive_
↪folder, or contact the author with the full output including the following error:\n",_
↪e)
```

In this tutorial, we work with the CIFAR10 dataset. In CIFAR10, each image has 3 color channels and is 32x32 pixels large. As autoencoders do not have the constrain of modeling images probabilistic, we can work on more complex image data (i.e. 3 color channels instead of black-and-white) much easier than for VAEs. In case you have downloaded CIFAR10 already in a different directory, make sure to set DATASET_PATH accordingly to prevent another download.

In contrast to previous tutorials on CIFAR10 like [Tutorial 5](#) (CNN classification), we do not normalize the data explicitly with a mean of 0 and std of 1, but roughly estimate it scaling the data between -1 and 1. This is because limiting the range will make our task of predicting/reconstructing images easier.

```
[3]: # Transformations applied on each image => only make them a tensor
transform = transforms.Compose([transforms.ToTensor(),
                               transforms.Normalize((0.5,), (0.5,))])

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=transform,
                        download=True)
pl.seed_everything(42)
train_set, val_set = torch.utils.data.random_split(train_dataset, [45000, 5000])

# Loading the test set
test_set = CIFAR10(root=DATASET_PATH, train=False, transform=transform, download=True)

# We define a set of data loaders that we can use for various purposes later.
train_loader = data.DataLoader(train_set, batch_size=256, shuffle=True, drop_last=True,
                                pin_memory=True, num_workers=4)
val_loader = data.DataLoader(val_set, batch_size=256, shuffle=False, drop_last=False,
                             num_workers=4)
test_loader = data.DataLoader(test_set, batch_size=256, shuffle=False, drop_last=False,
                             num_workers=4)

def get_train_images(num):
    return torch.stack([train_dataset[i][0] for i in range(num)], dim=0)
```

Files already downloaded and verified

(continues on next page)

(continued from previous page)

Files already downloaded and verified

4.12.1 Building the autoencoder

In general, an autoencoder consists of an **encoder** that maps the input x to a lower-dimensional feature vector z , and a **decoder** that reconstructs the input \hat{x} from z . We train the model by comparing x to \hat{x} and optimizing the parameters to increase the similarity between x and \hat{x} . See below for a small illustration of the autoencoder framework.

We first start by implementing the encoder. The encoder effectively consists of a deep convolutional network, where we scale down the image layer-by-layer using strided convolutions. After downscaling the image three times, we flatten the features and apply linear layers. The latent representation z is therefore a vector of size d which can be flexibly selected.

```
[4]: class Encoder(nn.Module):

    def __init__(self,
                 num_input_channels : int,
                 base_channel_size : int,
                 latent_dim : int,
                 act_fn : object = nn.GELU):
        """
        Inputs:
            - num_input_channels : Number of input channels of the image. For CIFAR, this parameter is 3
            - base_channel_size : Number of channels we use in the first convolutional layers. Deeper layers might use a duplicate of it.
            - latent_dim : Dimensionality of latent representation z
            - act_fn : Activation function used throughout the encoder network
        """
        super().__init__()
        c_hid = base_channel_size
        self.net = nn.Sequential(
            nn.Conv2d(num_input_channels, c_hid, kernel_size=3, padding=1, stride=2), # 32x32 => 16x16
            act_fn(),
            nn.Conv2d(c_hid, c_hid, kernel_size=3, padding=1),
            act_fn(),
            nn.Conv2d(c_hid, 2*c_hid, kernel_size=3, padding=1, stride=2), # 16x16 => 8x8
            act_fn(),
            nn.Conv2d(2*c_hid, 2*c_hid, kernel_size=3, padding=1),
            act_fn(),
            nn.Conv2d(2*c_hid, 2*c_hid, kernel_size=3, padding=1, stride=2), # 8x8 => 4x4
            act_fn(),
            nn.Flatten(), # Image grid to single feature vector
            nn.Linear(2*16*c_hid, latent_dim)
        )

    def forward(self, x):
        return self.net(x)
```

Note that we do not apply Batch Normalization here. This is because we want the encoding of each image to be independent of all the other images. Otherwise, we might introduce correlations into the encoding or decoding that we do not want to have. In some implementations, you still can see Batch Normalization being used, because it can also serve as a form of regularization. Nevertheless, the better practice is to go with other normalization techniques if necessary like Instance Normalization or Layer Normalization. Given the small size of the model, we can neglect normalization for now.

The decoder is a mirrored, flipped version of the encoder. The only difference is that we replace strided convolutions by transposed convolutions (i.e. deconvolutions) to upscale the features. Transposed convolutions can be imagined as adding the stride to the input instead of the output, and can thus upscale the input. For an illustration of a `nn.ConvTranspose2d` layer with kernel size 3, stride 2, and padding 1, see below (figure credit - [Vincent Dumoulin and Francesco Visin](#)):

You see that for an input of size 3×3 , we obtain an output of 5×5 . However, to truly have a reverse operation of the convolution, we need to ensure that the layer scales the input shape by a factor of 2 (e.g. $4 \times 4 \rightarrow 8 \times 8$). For this, we can specify the parameter `output_padding` which adds additional values to the output shape. Note that we do not perform zero-padding with this, but rather increase the output shape for calculation.

Overall, the decoder can be implemented as follows:

```
[5]: class Decoder(nn.Module):

    def __init__(self,
                 num_input_channels : int,
                 base_channel_size : int,
                 latent_dim : int,
                 act_fn : object = nn.GELU):
        """
        Inputs:
            - num_input_channels : Number of channels of the image to reconstruct. For
        ↵CIFAR, this parameter is 3
            - base_channel_size : Number of channels we use in the last convolutional
        ↵layers. Early layers might use a duplicate of it.
            - latent_dim : Dimensionality of latent representation z
            - act_fn : Activation function used throughout the decoder network
        """
        super().__init__()
        c_hid = base_channel_size
        self.linear = nn.Sequential(
            nn.Linear(latent_dim, 2*16*c_hid),
            act_fn()
        )
        self.net = nn.Sequential(
            nn.ConvTranspose2d(2*c_hid, 2*c_hid, kernel_size=3, output_padding=1, ↵
        ↵padding=1, stride=2), # 4x4 => 8x8
            act_fn(),
            nn.Conv2d(2*c_hid, 2*c_hid, kernel_size=3, padding=1),
            act_fn(),
            nn.ConvTranspose2d(2*c_hid, c_hid, kernel_size=3, output_padding=1, ↵
        ↵padding=1, stride=2), # 8x8 => 16x16
            act_fn(),
            nn.Conv2d(c_hid, c_hid, kernel_size=3, padding=1),
            act_fn(),
        )
```

(continues on next page)

(continued from previous page)

```

        nn.ConvTranspose2d(c_hid, num_input_channels, kernel_size=3, output_
    ↵padding=1, padding=1, stride=2), # 16x16 => 32x32
        nn.Tanh() # The input images is scaled between -1 and 1, hence the output_
    ↵has to be bounded as well
    )

    def forward(self, x):
        x = self.linear(x)
        x = x.reshape(x.shape[0], -1, 4, 4)
        x = self.net(x)
        return x

```

The encoder and decoder networks we chose here are relatively simple. Usually, more complex networks are applied, especially when using a ResNet-based architecture. For example, see [VQ-VAE](#) and [NVAE](#) (although the papers discuss architectures for VAEs, they can equally be applied to standard autoencoders).

In a final step, we add the encoder and decoder together into the autoencoder architecture. We define the autoencoder as PyTorch Lightning Module to simplify the needed training code:

```
[6]: class Autoencoder(pl.LightningModule):

    def __init__(self,
                 base_channel_size: int,
                 latent_dim: int,
                 encoder_class : object = Encoder,
                 decoder_class : object = Decoder,
                 num_input_channels: int = 3,
                 width: int = 32,
                 height: int = 32):
        super().__init__()
        # Saving hyperparameters of autoencoder
        self.save_hyperparameters()
        # Creating encoder and decoder
        self.encoder = encoder_class(num_input_channels, base_channel_size, latent_dim)
        self.decoder = decoder_class(num_input_channels, base_channel_size, latent_dim)
        # Example input array needed for visualizing the graph of the network
        self.example_input_array = torch.zeros(2, num_input_channels, width, height)

    def forward(self, x):
        """
        The forward function takes in an image and returns the reconstructed image
        """
        z = self.encoder(x)
        x_hat = self.decoder(z)
        return x_hat

    def _get_reconstruction_loss(self, batch):
        """
        Given a batch of images, this function returns the reconstruction loss (MSE in_
    ↵our case)
        """
        x, _ = batch # We do not need the labels
        x_hat = self.forward(x)

```

(continues on next page)

(continued from previous page)

```

loss = F.mse_loss(x, x_hat, reduction="none")
loss = loss.sum(dim=[1, 2, 3]).mean(dim=[0])
return loss

def configure_optimizers(self):
    optimizer = optim.Adam(self.parameters(), lr=1e-3)
    # Using a scheduler is optional but can be helpful.
    # The scheduler reduces the LR if the validation performance hasn't improved for
    # the last N epochs
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
                                                       mode='min',
                                                       factor=0.2,
                                                       patience=20,
                                                       min_lr=5e-5)
    return {"optimizer": optimizer, "lr_scheduler": scheduler, "monitor": "val_loss"}

def training_step(self, batch, batch_idx):
    loss = self._get_reconstruction_loss(batch)
    self.log('train_loss', loss)
    return loss

def validation_step(self, batch, batch_idx):
    loss = self._get_reconstruction_loss(batch)
    self.log('val_loss', loss)

def test_step(self, batch, batch_idx):
    loss = self._get_reconstruction_loss(batch)
    self.log('test_loss', loss)

```

For the loss function, we use the mean squared error (MSE). The mean squared error pushes the network to pay special attention to those pixel values its estimate is far away. Predicting 127 instead of 128 is not important when reconstructing, but confusing 0 with 128 is much worse. Note that in contrast to VAEs, we do not predict the probability per pixel value, but instead use a distance measure. This saves a lot of parameters and simplifies training. To get a better intuition per pixel, we report the summed squared error averaged over the batch dimension (any other mean/sum leads to the same result/parameters).

However, MSE has also some considerable disadvantages. Usually, MSE leads to blurry images where small noise/high-frequent patterns are removed as those cause a very low error. To ensure realistic images to be reconstructed, one could combine Generative Adversarial Networks (lecture 10) with autoencoders as done in several works (e.g. see [here](#), [here](#) or these [slides](#)). Additionally, comparing two images using MSE does not necessarily reflect their visual similarity. For instance, suppose the autoencoder reconstructs an image shifted by one pixel to the right and bottom. Although the images are almost identical, we can get a higher loss than predicting a constant pixel value for half of the image (see code below). An example solution for this issue includes using a separate, pre-trained CNN, and use a distance of visual features in lower layers as a distance measure instead of the original pixel-level comparison.

```
[7]: def compare_imgs(img1, img2, title_prefix=""):
    # Calculate MSE loss between both images
    loss = F.mse_loss(img1, img2, reduction="sum")
    # Plot images for visual comparison
    grid = torchvision.utils.make_grid(torch.stack([img1, img2], dim=0), nrow=2,
    # normalize=True, range=(-1,1))
    grid = grid.permute(1, 2, 0)
    plt.figure(figsize=(4,2))
```

(continues on next page)

(continued from previous page)

```
plt.title(f"{title_prefix} Loss: {loss.item():.2f}")
plt.imshow(grid)
plt.axis('off')
plt.show()

for i in range(2):
    # Load example image
    img, _ = train_dataset[i]
    img_mean = img.mean(dim=[1,2], keepdims=True)

    # Shift image by one pixel
    SHIFT = 1
    img_shifted = torch.roll(img, shifts=SHIFT, dims=1)
    img_shifted = torch.roll(img_shifted, shifts=SHIFT, dims=2)
    img_shifted[:, :, 1, :] = img_mean
    img_shifted[:, :, :, 1] = img_mean
    compare_imgs(img, img_shifted, "Shifted -")

    # Set half of the image to zero
    img_masked = img.clone()
    img_masked[:, :img_masked.shape[1]//2, :] = img_mean
    compare_imgs(img, img_masked, "Masked -")
```

Shifted - Loss: 205.40



Masked - Loss: 158.48



Shifted - Loss: 418.47



Masked - Loss: 295.20



Training the model

During the training, we want to keep track of the learning progress by seeing reconstructions made by our model. For this, we implement a callback object in PyTorch Lightning which will add reconstructions every N epochs to our tensorboard:

```
[8]: class GenerateCallback(pl.Callback):

    def __init__(self, input_imgs, every_n_epochs=1):
        super().__init__()
        self.input_imgs = input_imgs # Images to reconstruct during training
        self.every_n_epochs = every_n_epochs # Only save those images every N epochs
        # otherwise tensorboard gets quite large

    def on_train_epoch_end(self, trainer, pl_module):
        if trainer.current_epoch % self.every_n_epochs == 0:
            # Reconstruct images
            input_imgs = self.input_imgs.to(pl_module.device)
            with torch.no_grad():
                pl_module.eval()
                reconst_imgs = pl_module(input_imgs)
                pl_module.train()
            # Plot and add to tensorboard
            imgs = torch.stack([input_imgs, reconst_imgs], dim=1).flatten(0,1)
            grid = torchvision.utils.make_grid(imgs, nrow=2, normalize=True, range=(-1,
            1))
            trainer.logger.experiment.add_image("Reconstructions", grid, global_
            step=trainer.global_step)
```

(continues on next page)

(continued from previous page)

We will now write a training function that allows us to train the autoencoder with different latent dimensionality and returns both the test and validation score. We provide pre-trained models and recommend you using those, especially when you work on a computer without GPU. Of course, feel free to train your own models on Lisa.

```
[9]: def train_cifar(latent_dim):
    # Create a PyTorch Lightning trainer with the generation callback
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, f"cifar10_{latent_dim}"),
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=500,
                          callbacks=[ModelCheckpoint(save_weights_only=True),
                                     GenerateCallback(get_train_images(8), every_n_
                           epochs=10),
                                     LearningRateMonitor("epoch")])
    trainer.logger._log_graph = True          # If True, we plot the computation graph in_
                                         tensorboard
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't_
                                         need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, f"cifar10_{latent_dim}.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = Autoencoder.load_from_checkpoint(pretrained_filename)
    else:
        model = Autoencoder(base_channel_size=32, latent_dim=latent_dim)
        trainer.fit(model, train_loader, val_loader)
    # Test best model on validation and test set
    val_result = trainer.test(model, val_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"test": test_result, "val": val_result}
    return model, result
```

Comparing latent dimensionality

When training an autoencoder, we need to choose a dimensionality for the latent representation z . The higher the latent dimensionality, the better we expect the reconstruction to be. However, the idea of autoencoders is to *compress* data. Hence, we are also interested in keeping the dimensionality low. To find the best tradeoff, we can train multiple models with different latent dimensionalities. The original input has $32 \times 32 \times 3 = 3072$ pixels. Keeping this in mind, a reasonable choice for the latent dimensionality might be between 64 and 384:

```
[10]: model_dict = {}
for latent_dim in [64, 128, 256, 384]:
    model_ld, result_ld = train_cifar(latent_dim)
    model_dict[latent_dim] = {"model": model_ld, "result": result_ld}

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Found pretrained model, loading...
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',  
    ↴layout=Layout(flex='2'), max=...
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',  
    ↴layout=Layout(flex='2'), max=...
```

```
GPU available: True, used: True
```

```
WARNING: Logging before flag parsing goes to stderr.
```

```
I1123 10:35:57.297765 140189561788224 distributed.py:49] GPU available: True, used: True
```

```
TPU available: False, using: 0 TPU cores
```

```
I1123 10:35:57.298832 140189561788224 distributed.py:49] TPU available: False, using: 0  
    ↴TPU cores
```

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
I1123 10:35:57.299675 140189561788224 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_  
    ↴VISIBLE_DEVICES: [0]
```

```
Found pretrained model, loading...
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',  
    ↴layout=Layout(flex='2'), max=...
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',  
    ↴layout=Layout(flex='2'), max=...
```

```
GPU available: True, used: True
```

```
I1123 10:35:58.741801 140189561788224 distributed.py:49] GPU available: True, used: True
```

```
TPU available: False, using: 0 TPU cores
```

```
I1123 10:35:58.742980 140189561788224 distributed.py:49] TPU available: False, using: 0  
    ↴TPU cores
```

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
I1123 10:35:58.743852 140189561788224 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_  
    ↴VISIBLE_DEVICES: [0]
```

```
Found pretrained model, loading...
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',  
    ↴layout=Layout(flex='2'), max=...
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',  
    ↴layout=Layout(flex='2'), max=...
```

```
GPU available: True, used: True
```

```
I1123 10:36:00.166019 140189561788224 distributed.py:49] GPU available: True, used: True
```

```
TPU available: False, using: 0 TPU cores
```

```
I1123 10:36:00.167815 140189561788224 distributed.py:49] TPU available: False, using: 0  
    ↴TPU cores
```

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
I1123 10:36:00.169288 140189561788224 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_  
    ↴VISIBLE_DEVICES: [0]
```

```
Found pretrained model, loading...
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', ↴
    ↴layout=Layout(flex='2'), max=...
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', ↴
    ↴layout=Layout(flex='2'), max=...
```

After training the models, we can plot the reconstruction loss over the latent dimensionality to get an intuition how these two properties are correlated:

```
[11]: latent_dims = sorted([k for k in model_dict])
val_scores = [model_dict[k]["result"]["val"][@][["test_loss"]] for k in latent_dims]

fig = plt.figure(figsize=(6,4))
plt.plot(latent_dims, val_scores, '--', color="#000", marker="*", markeredgecolor="#000",
    ↴ markerfacecolor="y", markersize=16)
plt.xscale("log")
plt.xticks(latent_dims, labels=latent_dims)
plt.title("Reconstruction error over latent dimensionality", fontsize=14)
plt.xlabel("Latent dimensionality")
plt.ylabel("Reconstruction error")
plt.minorticks_off()
plt.ylim(0,100)
plt.show()
```



As we initially expected, the reconstruction loss goes down with increasing latent dimensionality. For our model and

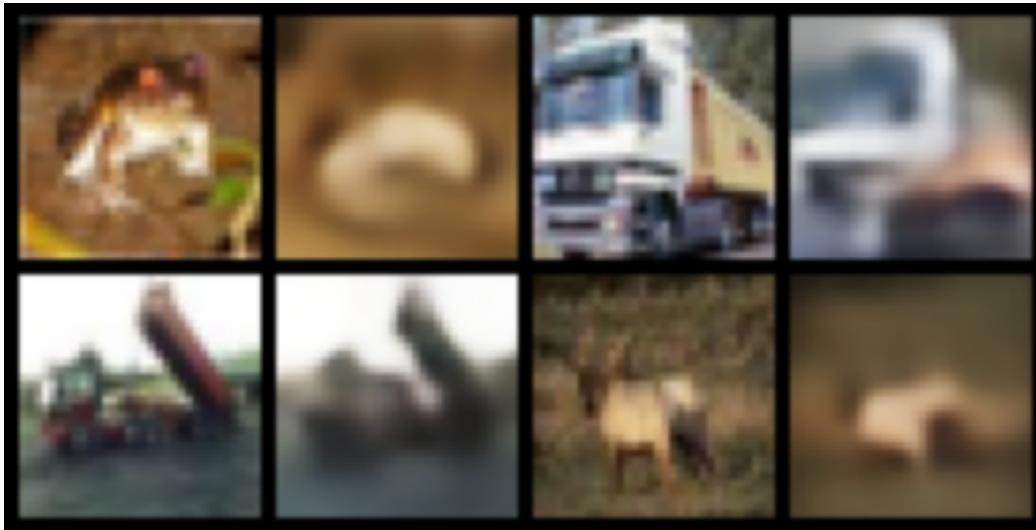
setup, the two properties seem to be exponentially (or double exponentially) correlated. To understand what these differences in reconstruction error mean, we can visualize example reconstructions of the four models. For simplicity, we visualize four training images of CIFAR10 we have seen already before. For larger models that may overfit, it is recommended to use images from the validation set.

```
[12]: def visualize_reconstructions(model, input_imgs):
    # Reconstruct images
    model.eval()
    with torch.no_grad():
        reconst_imgs = model(input_imgs.to(model.device))
    reconst_imgs = reconst_imgs.cpu()

    # Plotting
    imgs = torch.stack([input_imgs, reconst_imgs], dim=1).flatten(0,1)
    grid = torchvision.utils.make_grid(imgs, nrow=4, normalize=True, range=(-1,1))
    grid = grid.permute(1, 2, 0)
    plt.figure(figsize=(7,4.5))
    plt.title(f'Reconstructed from {model.hparams.latent_dim} latents')
    plt.imshow(grid)
    plt.axis('off')
    plt.show()
```

```
[13]: input_imgs = get_train_images(4)
for latent_dim in model_dict:
    visualize_reconstructions(model_dict[latent_dim]["model"], input_imgs)
```

Reconstructed from 64 latents

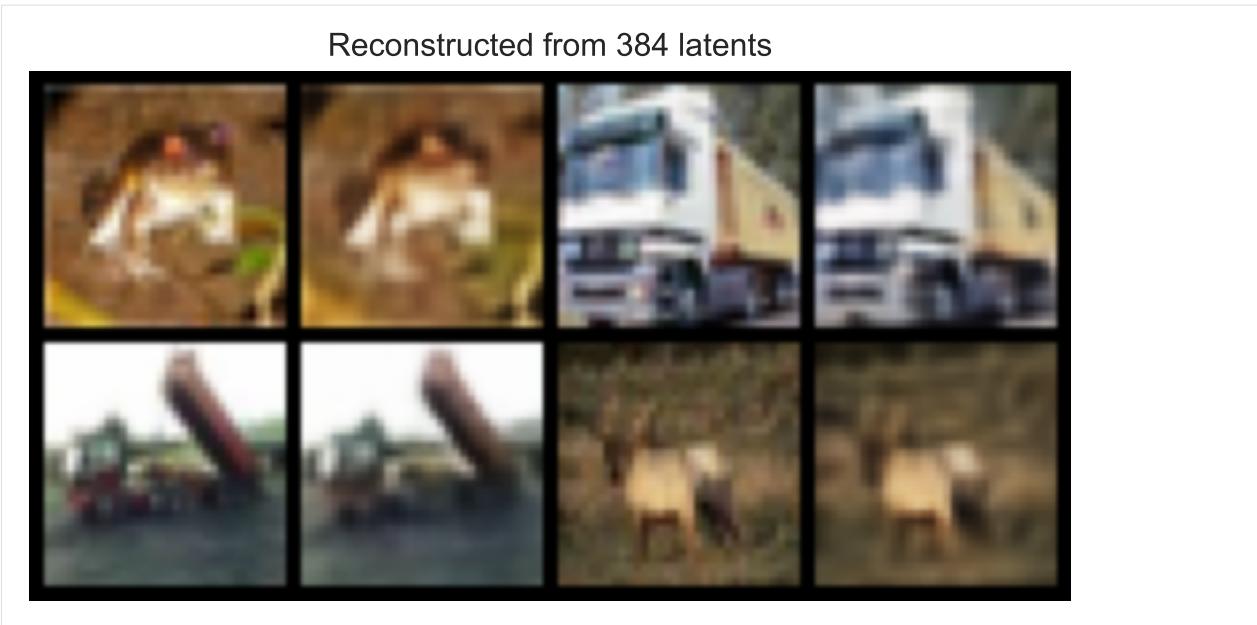


Reconstructed from 128 latents



Reconstructed from 256 latents





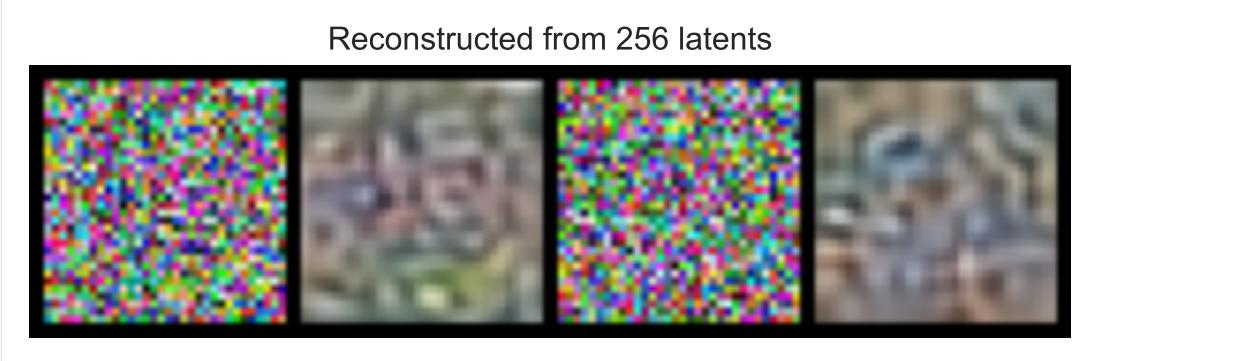
Clearly, the smallest latent dimensionality can only save information about the rough shape and color of the object, but the reconstructed image is extremely blurry and it is hard to recognize the original object in the reconstruction. With 128 features, we can recognize some shapes again although the picture remains blurry. The models with the highest two dimensionalities reconstruct the images quite well. The difference between 256 and 384 is marginal at first sight but can be noticed when comparing, for instance, the backgrounds of the first image (the 384 features model more of the pattern than 256).

Out-of-distribution images

Before continuing with the applications of autoencoder, we can actually explore some limitations of our autoencoder. For example, what happens if we try to reconstruct an image that is clearly out of the distribution of our dataset? We expect the decoder to have learned some common patterns in the dataset, and thus might in particular fail to reconstruct images that do not follow these patterns.

The first experiment we can try is to reconstruct noise. We, therefore, create two images whose pixels are randomly sampled from a uniform distribution over pixel values, and visualize the reconstruction of the model (feel free to test different latent dimensionalities):

```
[14]: rand_imgs = torch.rand(2, 3, 32, 32) * 2 - 1
visualize_reconstructions(model_dict[256]["model"], rand_imgs)
```



The reconstruction of the noise is quite poor, and seems to introduce some rough patterns. As the input does not follow

the patterns of the CIFAR dataset, the model has issues reconstructing it accurately.

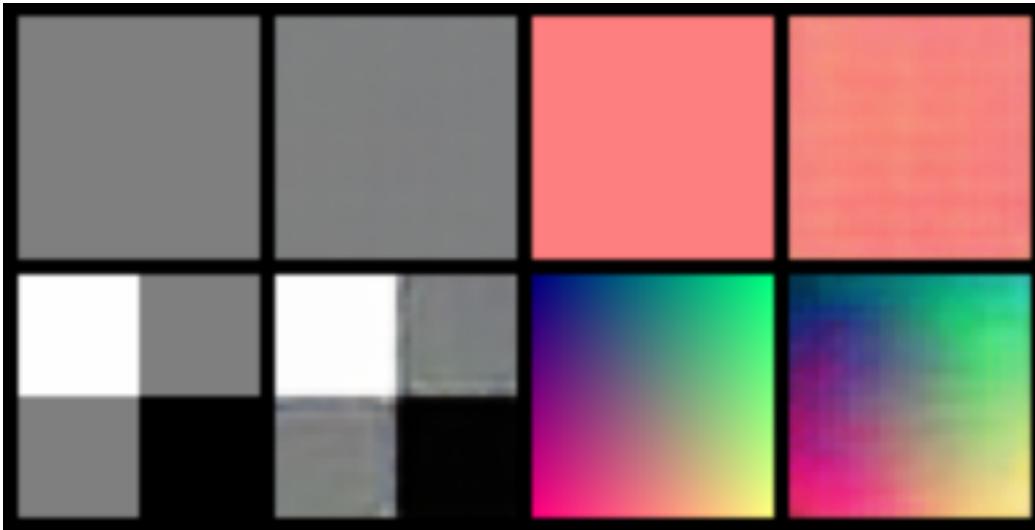
We can also check how well the model can reconstruct other manually-coded patterns:

```
[15]: plain_imgs = torch.zeros(4, 3, 32, 32)

# Single color channel
plain_imgs[1,0] = 1
# Checkboard pattern
plain_imgs[2,:,:16,:16] = 1
plain_imgs[2,:,:16,:16:] = -1
# Color progression
xx, yy = torch.meshgrid(torch.linspace(-1,1,32), torch.linspace(-1,1,32))
plain_imgs[3,0,:,:] = xx
plain_imgs[3,1,:,:] = yy

visualize_reconstructions(model_dict[256]["model"], plain_imgs)
```

Reconstructed from 256 latents



The plain, constant images are reconstructed relatively good although the single color channel contains some noticeable noise. The hard borders of the checkboard pattern are not as sharp as intended, as well as the color progression, both because such patterns never occur in the real-world pictures of CIFAR.

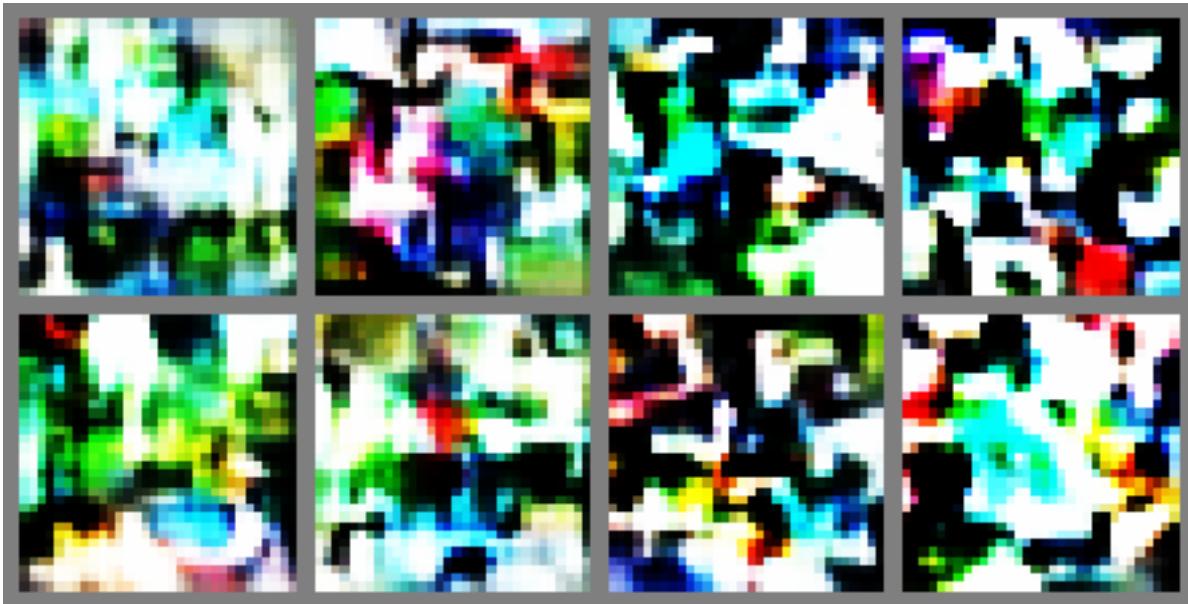
In general, autoencoders tend to fail reconstructing high-frequent noise (i.e. sudden, big changes across few pixels) due to the choice of MSE as loss function (see our previous discussion about loss functions in autoencoders). Small misalignments in the decoder can lead to huge losses so that the model settles for the expected value/mean in these regions. For low-frequent noise, a misalignment of a few pixels does not result in a big difference to the original image. However, the larger the latent dimensionality becomes, the more of this high-frequent noise can be accurately reconstructed.

Generating new images

Variational autoencoders are a generative version of the autoencoders because we regularize the latent space to follow a Gaussian distribution. However, in vanilla autoencoders, we do not have any restrictions on the latent vector. So what happens if we would actually input a randomly sampled latent vector into the decoder? Let's find it out below:

```
[16]: model = model_dict[256]["model"]
latent_vectors = torch.randn(8, model.hparams.latent_dim, device=model.device)
with torch.no_grad():
    imgs = model.decoder(latent_vectors)
    imgs = imgs.cpu()

grid = torchvision.utils.make_grid(imgs, nrow=4, normalize=True, range=(-1,1), pad_
    _value=0.5)
grid = grid.permute(1, 2, 0)
plt.figure(figsize=(8,5))
plt.imshow(grid)
plt.axis('off')
plt.show()
```



As we can see, the generated images more look like art than realistic images. As the autoencoder was allowed to structure the latent space in whichever way it suits the reconstruction best, there is no incentive to map every possible latent vector to realistic images. Furthermore, the distribution in latent space is unknown to us and doesn't necessarily follow a multivariate normal distribution. Thus, we can conclude that vanilla autoencoders are indeed not generative.

4.12.2 Finding visually similar images

One application of autoencoders is to build an image-based search engine to retrieve visually similar images. This can be done by representing all images as their latent dimensionality, and find the closest K images in this domain. The first step to such a search engine is to encode all images into z . In the following, we will use the training set as a search corpus, and the test set as queries to the system.

(Warning: the following cells can be computationally heavy for a weak CPU-only system. If you do not have a strong computer and are not on Google Colab, you might want to skip the execution of the following cells and rely on the results shown in the filled notebook)

```
[17]: # We use the following model throughout this section.
# If you want to try a different latent dimensionality, change it here!
model = model_dict[128]["model"]
```



```
[18]: def embed_imgs(model, data_loader):
    # Encode all images in the data_laoder using model, and return both images and
    # encodings
    img_list, embed_list = [], []
    model.eval()
    for imgs, _ in tqdm(data_loader, desc="Encoding images", leave=False):
        with torch.no_grad():
            z = model.encoder(imgs.to(model.device))
        img_list.append(imgs)
        embed_list.append(z)
    return (torch.cat(img_list, dim=0), torch.cat(embed_list, dim=0))

train_img_embeds = embed_imgs(model, train_loader)
test_img_embeds = embed_imgs(model, test_loader)

HBox(children=(FloatProgress(value=0.0, description='Encoding images', max=175.0,
    style=ProgressStyle(descript...
HBox(children=(FloatProgress(value=0.0, description='Encoding images', max=40.0,
    style=ProgressStyle(descripti...
```

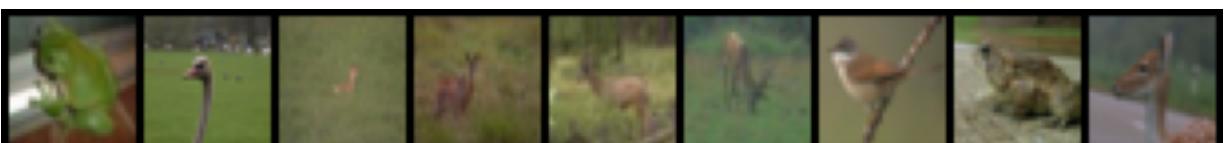
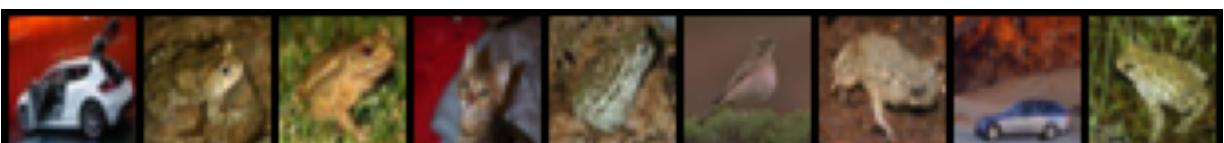
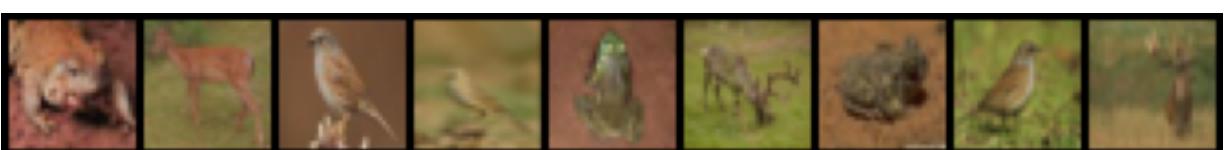
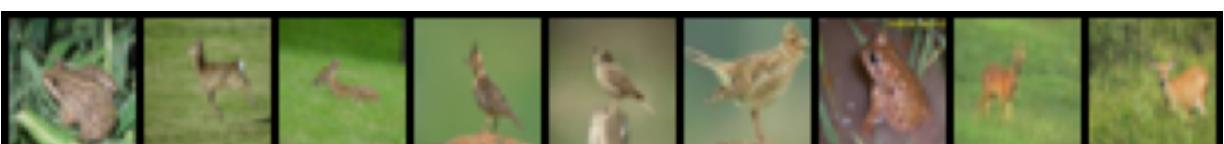
After encoding all images, we just need to write a function that finds the closest K images and returns (or plots) those:

```
[19]: def find_similar_images(query_img, query_z, key_embeds, K=8):
    # Find closest K images. We use the euclidean distance here but other like cosine
    # distance can also be used.
    dist = torch.cdist(query_z[None,:], key_embeds[1], p=2)
    dist = dist.squeeze(dim=0)
    dist, indices = torch.sort(dist)
    # Plot K closest images
    imgs_to_display = torch.cat([query_img[None], key_embeds[0][indices[:K]]], dim=0)
    grid = torchvision.utils.make_grid(imgs_to_display, nrow=K+1, normalize=True,
    range=(-1,1))
    grid = grid.permute(1, 2, 0)
    plt.figure(figsize=(12,3))
    plt.imshow(grid)
    plt.axis('off')
    plt.show()
```

```
[20]: # Plot the closest images for the first N test images as example
```

```
for i in range(8):
    find_similar_images(test_img_embeds[0][i], test_img_embeds[1][i], key_embeds=train_

```



Based on our autoencoder, we see that we are able to retrieve many similar images to the test input. In particular, in row 4, we can spot that some test images might not be that different from the training set as we thought (same poster, just

different scaling/color scaling). We also see that although we haven't given the model any labels, it can cluster different classes in different parts of the latent space (airplane + ship, animals, etc.). This is why autoencoders can also be used as a pre-training strategy for deep networks, especially when we have a large set of unlabeled images (often the case). However, it should be noted that the background still plays a big role in autoencoders while it doesn't for classification. Hence, we don't get "perfect" clusters and need to finetune such models for classification.

Tensorboard clustering

Another way of exploring the similarity of images in the latent space is by dimensionality-reduction methods like PCA or T-SNE. Luckily, Tensorboard provides a nice interface for this and we can make use of it in the following:

```
[21]: # We use the following model throughout this section.  
# If you want to try a different latent dimensionality, change it here!  
model = model_dict[128]["model"]
```

```
[22]: # Create a summary writer  
writer = SummaryWriter("tensorboard/")
```

The function `add_embedding` allows us to add high-dimensional feature vectors to TensorBoard on which we can perform clustering. What we have to provide in the function are the feature vectors, additional metadata such as the labels, and the original images so that we can identify a specific image in the clustering.

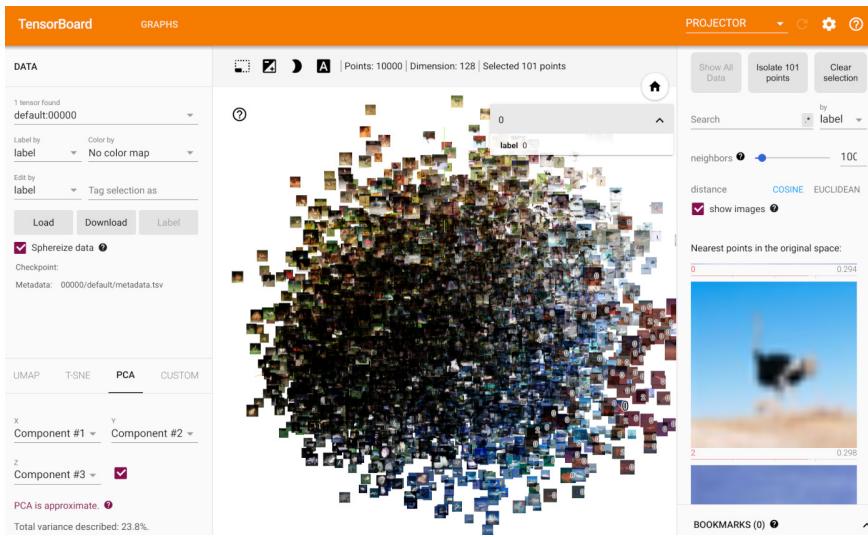
```
[23]: ## In case you obtain the following error in the next cell, execute the import  
→statements and last line in this cell  
## AttributeError: module 'tensorflow._api.v2.io.gfile' has no attribute 'get_filesystem'  
  
# import tensorflow as tf  
# import tensorboard as tb  
# tf.io.gfile = tb.compat.tensorflow_stub.io.gfile
```

```
[24]: # Note: the embedding projector in tensorboard is computationally heavy.  
# Reduce the image amount below if your computer struggles with visualizing all 10k  
→points  
NUM_IMGS = len(test_set)  
  
writer.add_embedding(test_img_embeds[1][:NUM_IMGS], # Encodings per image  
                     metadata=[test_set[i][1] for i in range(NUM_IMGS)], # Adding the  
→labels per image to the plot  
                     label_img=(test_img_embeds[0][:NUM_IMGS]+1)/2.0) # Adding the  
→original images to the plot
```

Finally, we can run tensorboard to explore similarities among images:

```
[25]: %tensorboard --logdir tensorboard/
```

You should be able to see something similar as in the following image. In case the projector stays empty, try to start the TensorBoard outside of the Jupyter notebook.



Overall, we can see that the model indeed clustered images together that are visually similar. Especially the background color seems to be a crucial factor in the encoding. This correlates to the chosen loss function, here Mean Squared Error on pixel-level because the background is responsible for more than half of the pixels in an average image. Hence, the model learns to focus on it. Nevertheless, we can see that the encodings also separate a couple of classes in the latent space although it hasn't seen any labels. This shows again that autoencoding can also be used as a "pre-training"/transfer learning task before classification.

```
[26]: # Closing the summary writer
writer.close()
```

4.12.3 Conclusion

In this tutorial, we have implemented our own autoencoder on small RGB images and explored various properties of the model. In contrast to variational autoencoders, vanilla AEs are not generative and can work on MSE loss functions. This makes them often easier to train. Both versions of AE can be used for dimensionality reduction, as we have seen for finding visually similar images beyond pixel distances. Despite autoencoders gaining less interest in the research community due to their more "theoretically" challenging counterpart of VAEs, autoencoders still find usage in a lot of applications like denoising and compression. Hence, AEs are an essential tool that every Deep Learning engineer/researcher should be familiar with.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.13 Tutorial 10: Adversarial attacks

Filled notebook:

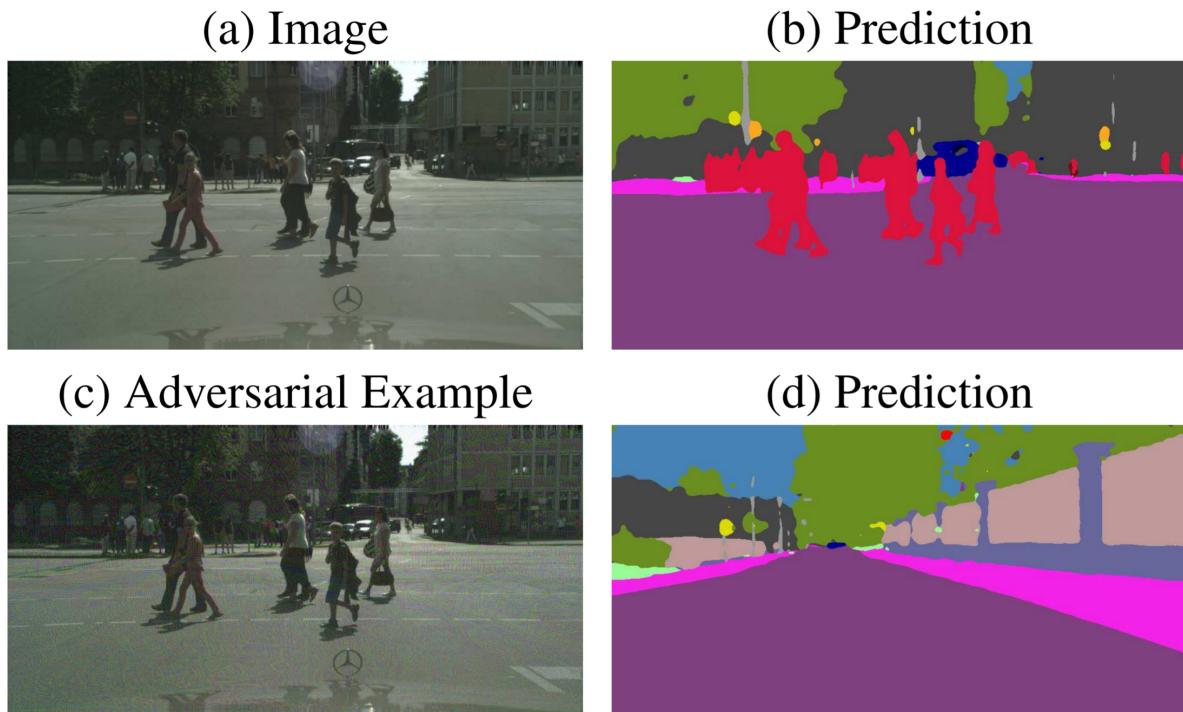
Pre-trained models and dataset:

Recordings:

Author: Phillip Lippe

In this tutorial, we will discuss adversarial attacks on deep image classification models. As we have seen in many of the previous tutorials so far, Deep Neural Networks are a very powerful tool to recognize patterns in data, and, for example, perform image classification on a human-level. However, we have not tested yet how robust these models actually are. Can we “trick” the model and find failure modes? Can we design images that the networks naturally classify incorrectly? Due to the high classification accuracy on unseen test data, we would expect that this can be difficult. However, in 2014, a research group at Google and NYU showed that deep CNNs can be easily fooled, just by adding some salient but carefully constructed noise to the images. For instance, take a look at the example below (figure credit - [Goodfellow et al.](#)):

The image on the left is the original image from ImageNet, and a deep CNN classifies the image correctly as “panda” with a class likelihood of 57%. Nevertheless, if we add a little noise to every pixel of the image, the prediction of the model changes completely. Instead of a panda, our CNN tells us that the image contains a “gibbon” with the confidence of over 99%. For a human, however, these two images look exactly alike, and you cannot distinguish which one has noise added and which doesn’t. While this first seems like a fun game to fool trained networks, it can have a serious impact on the usage of neural networks. More and more deep learning models are used in applications, such as for example autonomous driving. Imagine that someone who gains access to the camera input of the car, could make pedestrians “disappear” for the image understanding network by simply adding some noise to the input as shown below (the figure is taken from [J.H. Metzen et al.](#)). The first row shows the original image with the semantic segmentation output on the right (pedestrians red), while the second row shows the image with small noise and the corresponding segmentation prediction. The pedestrian becomes invisible for the network, and the car would think the road is clear ahead.



Some attack types don't even require to add noise, but minor changes on a stop sign can be already sufficient for the network to recognize it as a "50km/h" speed sign ([paper](#), [paper](#)). The consequences of such attacks can be devastating. Hence, every deep learning engineer who designs models for an application should be aware of the possibility of adversarial attacks.

To understand what makes CNNs vulnerable to such attacks, we will implement our own adversarial attack strategies in this notebook, and try to fool a deep neural network. Let's begin with importing our standard libraries:

```
[1]: ## Standard libraries
import os
import json
import math
import time
import numpy as np
import scipy.linalg

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.set()

## Progress bar
from tqdm.notebook import tqdm
```

(continues on next page)

(continued from previous page)

```

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
# Torchvision
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms
# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial10"

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Fetching the device that will be used throughout this notebook
device = torch.device("cpu") if not torch.cuda.is_available() else torch.device("cuda:0")
print("Using device", device)

Using device cuda:0

```

We have again a few download statements. This includes both a dataset, and a few pretrained patches we will use later.

```

[2]: import urllib.request
from urllib.error import HTTPError
import zipfile
# Github URL where the dataset is stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial10/"
# Files to download
pretrained_files = [(DATASET_PATH, "TinyImageNet.zip"), (CHECKPOINT_PATH, "patches.zip")]
# Create checkpoint path if it doesn't exist yet
os.makedirs(DATASET_PATH, exist_ok=True)
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for dir_name, file_name in pretrained_files:
    file_path = os.path.join(dir_name, file_name)

```

(continues on next page)

(continued from previous page)

```

if not os.path.isfile(file_path):
    file_url = base_url + file_name
    print(f"Downloading {file_url}...")
try:
    urllib.request.urlretrieve(file_url, file_path)
except HTTPError as e:
    print("Something went wrong. Please try to download the file from the GDrive_"
    "folder, or contact the author with the full output including the following error:\n",
    e)
if file_name.endswith(".zip"):
    print("Unzipping file...")
    with zipfile.ZipFile(file_path, 'r') as zip_ref:
        zip_ref.extractall(file_path.rsplit("/", 1)[0])

```

4.13.1 Deep CNNs on ImageNet

For our experiments in this notebook, we will use common CNN architectures trained on the ImageNet dataset. Such models are luckily provided by PyTorch's torchvision package, and hence we just need to load the model of our preference. For the results on the website and default on Google Colab, we use a ResNet34. Feel free to experiment with other architectures as well, the code is mainly independent of the specific architecture we choose.

```
[3]: # Load CNN architecture pretrained on ImageNet
os.environ["TORCH_HOME"] = CHECKPOINT_PATH
pretrained_model = torchvision.models.resnet34(weights='IMAGENET1K_V1')
pretrained_model = pretrained_model.to(device)

# No gradients needed for the network
pretrained_model.eval()
for p in pretrained_model.parameters():
    p.requires_grad = False
```

To perform adversarial attacks, we also need a dataset to work on. Given that the CNN model has been trained on ImageNet, it is only fair to perform the attacks on data from ImageNet. For this, we provide a small set of pre-processed images from the original ImageNet dataset (note that this dataset is shared under the same license as the original ImageNet dataset). Specifically, we have 5 images for each of the 1000 labels of the dataset. We can load the data below, and create a corresponding data loader.

```
[4]: # Mean and Std from ImageNet
NORM_MEAN = np.array([0.485, 0.456, 0.406])
NORM_STD = np.array([0.229, 0.224, 0.225])
# No resizing and center crop necessary as images are already preprocessed.
plain_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=NORM_MEAN,
                        std=NORM_STD)
])

# Load dataset and create data loader
imagenet_path = os.path.join(DATASET_PATH, "TinyImageNet/")
assert os.path.isdir(imagenet_path), f"Could not find the ImageNet dataset at expected_"
"path \\"{imagenet_path}\\". " + \
```

(continues on next page)

(continued from previous page)

```

f"Please make sure to have downloaded the ImageNet_"
→dataset here, or change the {DATASET_PATH=} variable."
dataset = torchvision.datasets.ImageFolder(root=imagenet_path, transform=plain_
→transforms)
data_loader = data.DataLoader(dataset, batch_size=32, shuffle=False, drop_last=False,_
→num_workers=8)

# Load label names to interpret the label numbers 0 to 999
with open(os.path.join(imagenet_path, "label_list.json"), "r") as f:
    label_names = json.load(f)

def get_label_index(lab_str):
    assert lab_str in label_names, f"Label \\"{lab_str}\\" not found. Check the spelling_"
→of the class."
    return label_names.index(lab_str)

```

Before we start with our attacks, we should verify the performance of our model. As ImageNet has 1000 classes, simply looking at the accuracy is not sufficient to tell the performance of a model. Imagine a model that always predicts the true label as the second-highest class in its softmax output. Although we would say it recognizes the object in the image, it achieves an accuracy of 0. In ImageNet with 1000 classes, there is not always one clear label we can assign an image to. This is why for image classifications over so many classes, a common alternative metric is “Top-5 accuracy”, which tells us how many times the true label has been within the 5 most-likely predictions of the model. As models usually perform quite well on those, we report the error (1 - accuracy) instead of the accuracy:

```
[5]: def eval_model(dataset_loader, img_func=None):
    tp, tp_5, counter = 0., 0., 0.
    for imgs, labels in tqdm(dataset_loader, desc="Validating..."):
        imgs = imgs.to(device)
        labels = labels.to(device)
        if img_func is not None:
            imgs = img_func(imgs, labels)
        with torch.no_grad():
            preds = pretrained_model(imgs)
            tp += (preds.argmax(dim=-1) == labels).sum()
            tp_5 += (preds.topk(5, dim=-1)[1] == labels[..., None]).any(dim=-1).sum()
            counter += preds.shape[0]
    acc = tp.float().item() / counter
    top5 = tp_5.float().item() / counter
    print(f"Top-1 error: {(100.0 * (1 - acc)):.2f}%")
    print(f"Top-5 error: {(100.0 * (1 - top5)):.2f}%")
    return acc, top5
```

```
[6]: _ = eval_model(data_loader)
HBox(children=(FloatProgress(value=0.0, description='Validating...', max=157.0,_
→style=ProgressStyle(descriptio...))

Top-1 error: 19.10%
Top-5 error: 4.30%
```

The ResNet34 achieves a decent error rate of 4.3% for the top-5 predictions. Next, we can look at some predictions of the model to get more familiar with the dataset. The function below plots an image along with a bar diagram of its predictions. We also prepare it to show adversarial examples for later applications.

```
[7]: def show_prediction(img, label, pred, K=5, adv_img=None, noise=None):

    if isinstance(img, torch.Tensor):
        # Tensor image to numpy
        img = img.cpu().permute(1, 2, 0).numpy()
        img = (img * NORM_STD[None, None]) + NORM_MEAN[None, None]
        img = np.clip(img, a_min=0.0, a_max=1.0)
        label = label.item()

        # Plot on the left the image with the true label as title.
        # On the right, have a horizontal bar plot with the top k predictions including
        ↪probabilities
        if noise is None or adv_img is None:
            fig, ax = plt.subplots(1, 2, figsize=(10, 2), gridspec_kw={'width_ratios': [1, 1]})
        ↪)
        else:
            fig, ax = plt.subplots(1, 5, figsize=(12, 2), gridspec_kw={'width_ratios': [1, 1, ↪
            ↪1, 1, 2]})

        ax[0].imshow(img)
        ax[0].set_title(label_names[label])
        ax[0].axis('off')

        if adv_img is not None and noise is not None:
            # Visualize adversarial images
            adv_img = adv_img.cpu().permute(1, 2, 0).numpy()
            adv_img = (adv_img * NORM_STD[None, None]) + NORM_MEAN[None, None]
            adv_img = np.clip(adv_img, a_min=0.0, a_max=1.0)
            ax[1].imshow(adv_img)
            ax[1].set_title('Adversarial')
            ax[1].axis('off')
            # Visualize noise
            noise = noise.cpu().permute(1, 2, 0).numpy()
            noise = noise * 0.5 + 0.5 # Scale between 0 to 1
            ax[2].imshow(noise)
            ax[2].set_title('Noise')
            ax[2].axis('off')
            # buffer
            ax[3].axis('off')

        if abs(pred.sum().item() - 1.0) > 1e-4:
            pred = torch.softmax(pred, dim=-1)
            topk_vals, topk_idx = pred.topk(K, dim=-1)
            topk_vals, topk_idx = topk_vals.cpu().numpy(), topk_idx.cpu().numpy()
            ax[-1].barh(np.arange(K), topk_vals*100.0, align='center', color=[["C0" if topk_
            ↪idx[i]==label else "C2" for i in range(K)]])
            ax[-1].set_yticks(np.arange(K))
            ax[-1].set_yticklabels([label_names[c] for c in topk_idx])
            ax[-1].invert_yaxis()
            ax[-1].set_xlabel('Confidence')
            ax[-1].set_title('Predictions')

    plt.show()
```

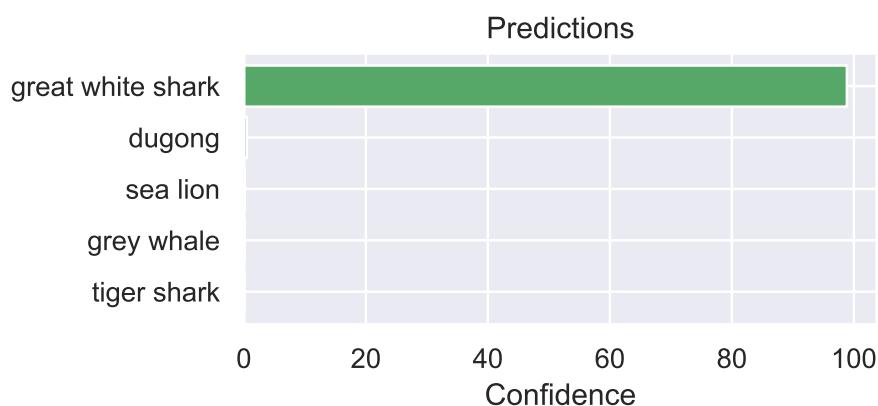
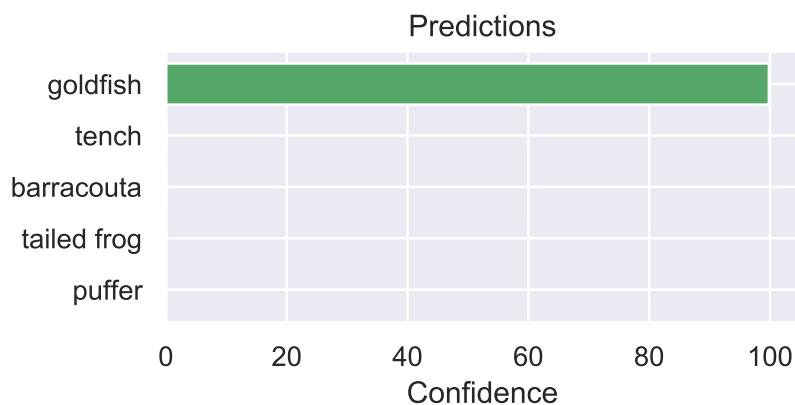
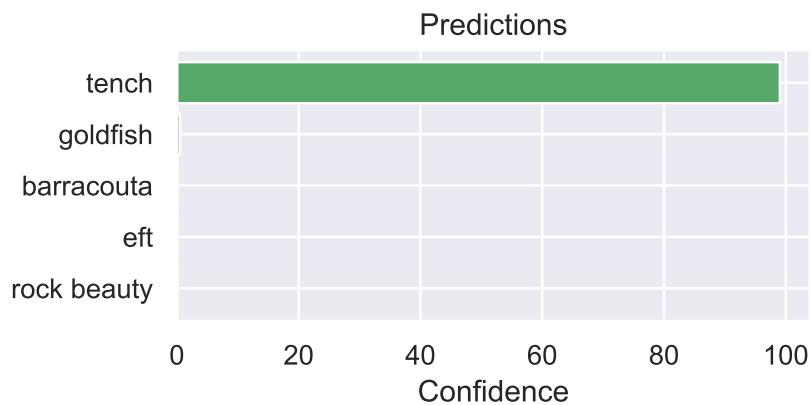
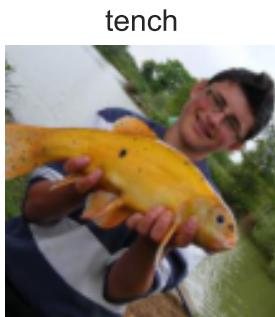
(continues on next page)

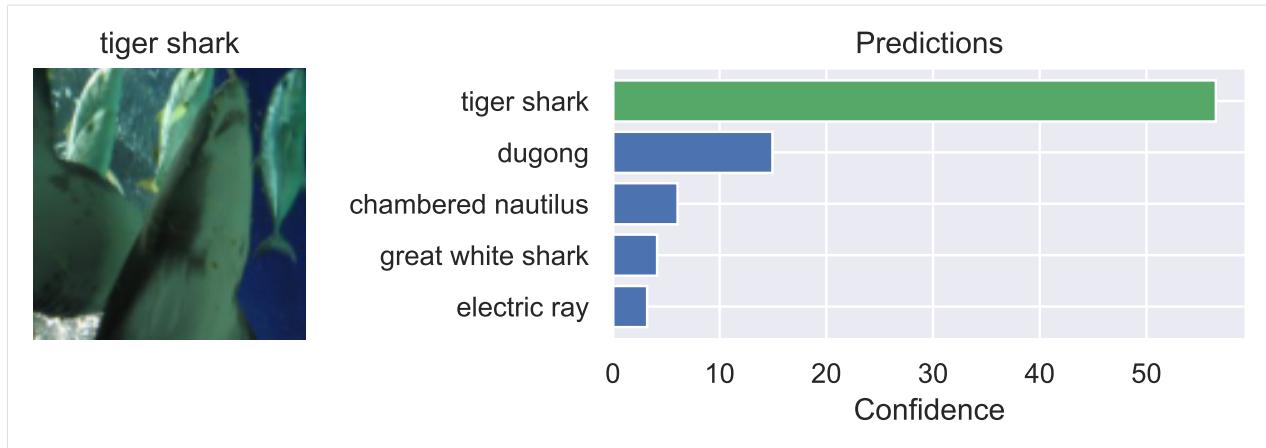
(continued from previous page)

```
plt.close()
```

Let's visualize a few images below:

```
[8]: exmp_batch, label_batch = next(iter(data_loader))
with torch.no_grad():
    preds = pretrained_model(exmp_batch.to(device))
for i in range(1,17,5):
    show_prediction(exmp_batch[i], label_batch[i], preds[i])
```





The bar plot on the right shows the top-5 predictions of the model with their class probabilities. We denote the class probabilities with “confidence” as it somewhat resembles how confident the network is that the image is of one specific class. Some of the images have a highly peaked probability distribution, and we would expect the model to be rather robust against noise for those. However, we will see below that this is not always the case. Note that all of the images are of fish because the data loader doesn’t shuffle the dataset. Otherwise, we would get different images every time we run the notebook, which would make it hard to discuss the results on the static version.

4.13.2 White-box adversarial attacks

There have been proposed many possible adversarial attack strategies, which all share the same goal: alternate the data/image input only a little bit to have a great impact on the model’s prediction. Specifically, if we look at the ImageNet predictions above, how can we have to change the image of the goldfish so that the model does not recognize the goldfish anymore? At the same time, the label of the image should not change, in the sense that a human would still clearly classify it as a goldfish. This is the same objective that the generator network has in the Generative Adversarial Network framework: try to fool another network (discriminator) by changing its input.

Adversarial attacks are usually grouped into “white-box” and “black-box” attacks. White-box attacks assume that we have access to the model parameter and can, for example, calculate the gradients with respect to the input (similar as in GANs). Black-box attacks on the other hand have the harder task of not having any knowledge about the network, and can only obtain predictions for an image, but no gradients or the like. In this notebook, we will focus on white-box attacks as they are usually easier to implement and follow the intuition of Generative Adversarial Networks (GAN) as studied in lecture 10.

Fast Gradient Sign Method (FGSM)

One of the first attack strategies proposed is Fast Gradient Sign Method (FGSM), developed by Ian Goodfellow et al. in 2014. Given an image, we create an adversarial example by the following expression:

$$\tilde{x} = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$$

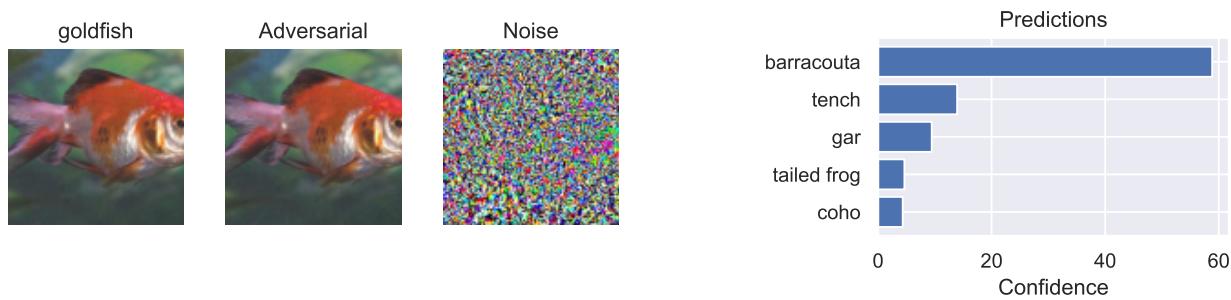
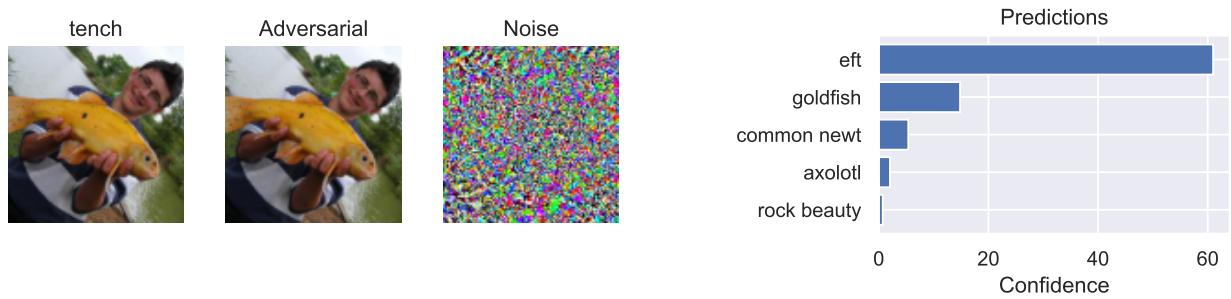
The term $J(\theta, x, y)$ represents the loss of the network for classifying input image x as label y ; ϵ is the intensity of the noise, and \tilde{x} the final adversarial example. The equation resembles SGD and is actually nothing else than that. We change the input image x in the direction of *maximizing* the loss $J(\theta, x, y)$. This is exactly the other way round as during training, where we try to minimize the loss. The sign function and ϵ can be seen as gradient clipping and learning rate specifically. We only allow our attack to change each pixel value by ϵ . You can also see that the attack can be performed very fast, as it only requires a single forward and backward pass. Let’s implement it below:

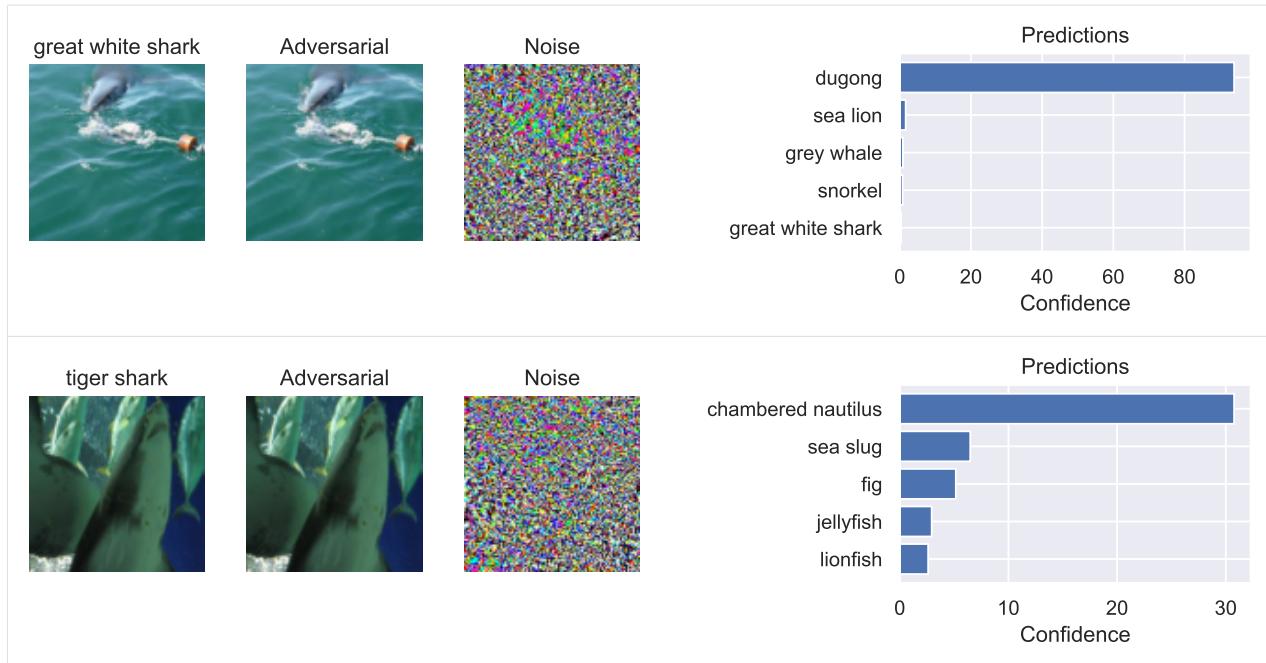
```
[9]: def fast_gradient_sign_method(model, imgs, labels, epsilon=0.02):
    # Determine prediction of the model
    inp_imgs = imgs.clone().requires_grad_()
    preds = model(inp_imgs.to(device))
    preds = F.log_softmax(preds, dim=-1)
    # Calculate loss by NLL
    loss = -torch.gather(preds, 1, labels.to(device).unsqueeze(dim=-1))
    loss.sum().backward()
    # Update image to adversarial example as written above
    noise_grad = torch.sign(inp_imgs.grad.to(imgs.device))
    fake_imgs = imgs + epsilon * noise_grad
    fake_imgs.detach_()
    return fake_imgs, noise_grad
```

The default value of $\epsilon = 0.02$ corresponds to changing a pixel value by about 1 in the range of 0 to 255, e.g. changing 127 to 128. This difference is marginal and can often not be recognized by humans. Let's try it below on our example images:

```
[10]: adv_imgs, noise_grad = fast_gradient_sign_method(pretrained_model, exmp_batch, label_
         →batch, epsilon=0.02)
      with torch.no_grad():
          adv_preds = pretrained_model(adv_imgs.to(device))

      for i in range(1,17,5):
          show_prediction(exmp_batch[i], label_batch[i], adv_preds[i], adv_img=adv_imgs[i], ↴
                           →noise=noise_grad[i])
```





Despite the minor amount of noise, we are able to fool the network on all of our examples. None of the labels have made it into the top-5 for the four images, showing that we indeed fooled the model. We can also check the accuracy of the model on the adversarial images:

```
[11]: _ = eval_model(data_loader, img_func=lambda x, y: fast_gradient_sign_method(pretrained_
    ↵model, x, y, epsilon=0.02)[0])
HBox(children=(FloatProgress(value=0.0, description='Validating...', max=157.0, ↵
    ↵style=ProgressStyle(descriptio...
```

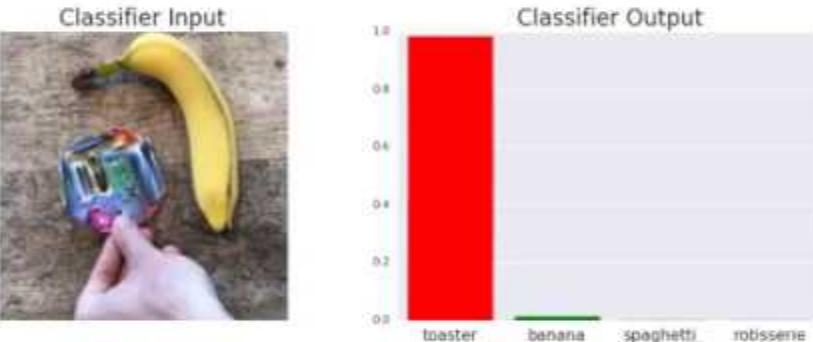
Top-1 error: 93.56%

Top-5 error: 60.52%

As expected, the model is fooled on almost every image at least for the top-1 error, and more than half don't have the true label in their top-5. This is a quite significant difference compared to the error rate of 4.3% on the clean images. However, note that the predictions remain semantically similar. For instance, in the images we visualized above, the tench is still recognized as another fish, as well as the great white shark being a dugong. FGSM could be adapted to increase the probability of a specific class instead of minimizing the probability of a label, but for those, there are usually better attacks such as the adversarial patch.

Adversarial Patches

Instead of changing every pixel by a little bit, we can also try to change a small part of the image into whatever values we would like. In other words, we will create a small image patch that covers a minor part of the original image but causes the model to confidently predict a specific class we choose. This form of attack is an even bigger threat in real-world applications than FSGM. Imagine a network in an autonomous car that receives a live image from a camera. Another driver could print out a specific pattern and put it on the back of his/her vehicle to make the autonomous car believe that the car is actually a pedestrian. Meanwhile, humans would not notice it. Tom Brown et al. proposed a way of learning such adversarial image patches robustly in 2017 and provided a short demonstration on YouTube. Interestingly, if you add a small picture of the target class (here *toaster*) to the original image, the model does not pick it up at all. A specifically designed patch, however, which only roughly looks like a toaster, can change the network's prediction instantaneously.



Let's take a closer look at how we can actually train such patches. The general idea is very similar to FSGM in the sense that we calculate gradients for the input, and update our adversarial input correspondingly. However, there are also some differences in the setup. Firstly, we do not calculate a gradient for every pixel. Instead, we replace parts of the input image with our patch and then calculate the gradients just for our patch. Secondly, we don't just do it for one image, but we want the patch to work with any possible image. Hence, we have a whole training loop where we train the patch using SGD. Lastly, image patches are usually designed to make the model predict a specific class, not just any other arbitrary class except the true label. For instance, we can try to create a patch for the class "toaster" and train the patch so that our pretrained model predicts the class "toaster" for any image with the patch in it.

Additionally, to the setup described above, there are a couple of design choices we can take. For instance, Brown et al. randomly rotated and scaled the patch during training before placing it at a random position in an input image. This makes the patch more robust to small changes and is necessary if we want to fool a neural network in a real-world application. For simplicity, we will only focus on making the patch robust to the location in the image. Given a batch of input images and a patch, we can add the patch as follows:

```
[12]: def place_patch(img, patch):
    for i in range(img.shape[0]):
        h_offset = np.random.randint(0, img.shape[2]-patch.shape[1]-1)
        w_offset = np.random.randint(0, img.shape[3]-patch.shape[2]-1)
        img[i,:,h_offset:h_offset+patch.shape[1],w_offset:w_offset+patch.shape[2]] =_
        patch_forward(patch)
    return img
```

The patch itself will be an `nn.Parameter` whose values are in the range between $-\infty$ and ∞ . Images are, however, naturally limited in their range, and thus we write a small function that maps the parameter into the image value range of ImageNet:

```
[13]: TENSOR_MEANS, TENSOR_STD = torch.FloatTensor(NORM_MEAN)[:,None,None], torch.-
        FloatTensor(NORM_STD)[:,None,None]
def patch_forward(patch):
    # Map patch values from [-infty,infty] to ImageNet min and max
```

(continues on next page)

(continued from previous page)

```
patch = (torch.tanh(patch) + 1 - 2 * TENSOR_MEANS) / (2 * TENSOR_STD)
return patch
```

Before looking at the actual training code, we can write a small evaluation function. We evaluate the success of a patch by how many times we were able to fool the network into predicting our target class. A simple function for this is implemented below.

```
[14]: def eval_patch(model, patch, val_loader, target_class):
    model.eval()
    tp, tp_5, counter = 0., 0., 0.
    with torch.no_grad():
        for img, img_labels in tqdm(val_loader, desc="Validating...", leave=False):
            # For stability, place the patch at 4 random locations per image, and
            # average the performance
            for _ in range(4):
                patch_img = place_patch(img, patch)
                patch_img = patch_img.to(device)
                img_labels = img_labels.to(device)
                pred = model(patch_img)
                # In the accuracy calculation, we need to exclude the images that are of
                # our target class
                # as we would not "fool" the model into predicting those
                tp += torch.logical_and(pred.argmax(dim=-1) == target_class, img_labels !=
                target_class).sum()
                tp_5 += torch.logical_and((pred.topk(5, dim=-1)[1] == target_class).
                any(dim=-1), img_labels != target_class).sum()
                counter += (img_labels != target_class).sum()
            acc = tp/counter
            top5 = tp_5/counter
    return acc, top5
```

Finally, we can look at the training loop. Given a model to fool, a target class to design the patch for, and a size k of the patch in the number of pixels, we first start by creating a parameter of size $3 \times k \times k$. These are the only parameters we will train, and the network itself remains untouched. We use a simple SGD optimizer with momentum to minimize the classification loss of the model given the patch in the image. While we first start with a very high loss due to the good initial performance of the network, the loss quickly decreases once we start changing the patch. In the end, the patch will represent patterns that are characteristic of the class. For instance, if we would want the model to predict a “goldfish” in every image, we would expect the pattern to look somewhat like a goldfish. Over the iterations, the model finetunes the pattern and, hopefully, achieves a high fooling accuracy.

```
[15]: def patch_attack(model, target_class, patch_size=64, num_epochs=5):
    # Leave a small set of images out to check generalization
    # In most of our experiments, the performance on the hold-out data points
    # was as good as on the training set. Overfitting was little possible due
    # to the small size of the patches.
    train_set, val_set = torch.utils.data.random_split(dataset, [4500, 500])
    train_loader = data.DataLoader(train_set, batch_size=32, shuffle=True, drop_
    last=True, num_workers=8)
    val_loader = data.DataLoader(val_set, batch_size=32, shuffle=False, drop_last=False,
    num_workers=4)

    # Create parameter and optimizer
```

(continues on next page)

(continued from previous page)

```

if not isinstance(patch_size, tuple):
    patch_size = (patch_size, patch_size)
patch = nn.Parameter(torch.zeros(3, patch_size[0], patch_size[1]), requires_
→grad=True)
optimizer = torch.optim.SGD([patch], lr=1e-1, momentum=0.8)
loss_module = nn.CrossEntropyLoss()

# Training loop
for epoch in range(num_epochs):
    t = tqdm(train_loader, leave=False)
    for img, _ in t:
        img = place_patch(img, patch)
        img = img.to(device)
        pred = model(img)
        labels = torch.zeros(img.shape[0], device=pred.device, dtype=torch.long) .
→fill_(target_class)
        loss = loss_module(pred, labels)
        optimizer.zero_grad()
        loss.mean().backward()
        optimizer.step()
        t.set_description(f"Epoch {epoch}, Loss: {loss.item():4.2f}")

# Final validation
acc, top5 = eval_patch(model, patch, val_loader, target_class)

return patch.data, {"acc": acc.item(), "top5": top5.item()}

```

To get some experience with what to expect from an adversarial patch attack, we want to train multiple patches for different classes. As the training of a patch can take one or two minutes on a GPU, we have provided a couple of pre-trained patches including their results on the full dataset. The results are saved in a JSON file, which is loaded below.

```

[16]: # Load evaluation results of the pretrained patches
json_results_file = os.path.join(CHECKPOINT_PATH, "patch_results.json")
json_results = []
if os.path.isfile(json_results_file):
    with open(json_results_file, "r") as f:
        json_results = json.load(f)

# If you train new patches, you can save the results via calling this function
def save_results(patch_dict):
    result_dict = {cname: {psize: [t.item() if isinstance(t, torch.Tensor) else t
                                    for t in patch_dict[cname][psize]["results"]]
                           for psize in patch_dict[cname]} for cname in patch_dict}
    with open(os.path.join(CHECKPOINT_PATH, "patch_results.json"), "w") as f:
        json.dump(result_dict, f, indent=4)

```

Additionally, we implement a function to train and evaluate patches for a list of classes and patch sizes. The pretrained patches include the classes *toaster*, *goldfish*, *school bus*, *lipstick*, and *pineapple*. We chose the classes arbitrarily to cover multiple domains (animals, vehicles, fruits, devices, etc.). We trained each class for three different patch sizes: 32×32 pixels, 48×48 pixels, and 64×64 pixels. We can load them in the two cells below.

```
[17]: def get_patches(class_names, patch_sizes):
    result_dict = dict()

    # Loop over all classes and patch sizes
    for name in class_names:
        result_dict[name] = dict()
        for patch_size in patch_sizes:
            c = label_names.index(name)
            file_name = os.path.join(CHECKPOINT_PATH, f"{name}_{patch_size}_patch.pt")
            # Load patch if pretrained file exists, otherwise start training
            if not os.path.isfile(file_name):
                patch, val_results = patch_attack(pretrained_model, target_class=c,
                patch_size=patch_size, num_epochs=5)
                print(f"Validation results for {name} and {patch_size}:", val_results)
                torch.save(patch, file_name)
            else:
                patch = torch.load(file_name)
            # Load evaluation results if exist, otherwise manually evaluate the patch
            if name in json_results:
                results = json_results[name][str(patch_size)]
            else:
                results = eval_patch(pretrained_model, patch, data_loader, target_
                class=c)

            # Store results and the patches in a dict for better access
            result_dict[name][patch_size] = {
                "results": results,
                "patch": patch
            }

    return result_dict
```

Feel free to add any additional classes and/or patch sizes.

```
[18]: class_names = ['toaster', 'goldfish', 'school bus', 'lipstick', 'pineapple']
patch_sizes = [32, 48, 64]

patch_dict = get_patches(class_names, patch_sizes)
# save_results(patch_dict) # Uncomment if you add new class names and want to save the_
# new results
```

Before looking at the quantitative results, we can actually visualize the patches.

```
[19]: def show_patches():
    fig, ax = plt.subplots(len(patch_sizes), len(class_names), figsize=(len(class_
    names)*2.2, len(patch_sizes)*2.2))
    for c_idx, cname in enumerate(class_names):
        for p_idx, psze in enumerate(patch_sizes):
            patch = patch_dict[cname][psze]["patch"]
            patch = (torch.tanh(patch) + 1) / 2 # Parameter to pixel values
            patch = patch.cpu().permute(1, 2, 0).numpy()
            patch = np.clip(patch, a_min=0.0, a_max=1.0)
            ax[p_idx][c_idx].imshow(patch)
```

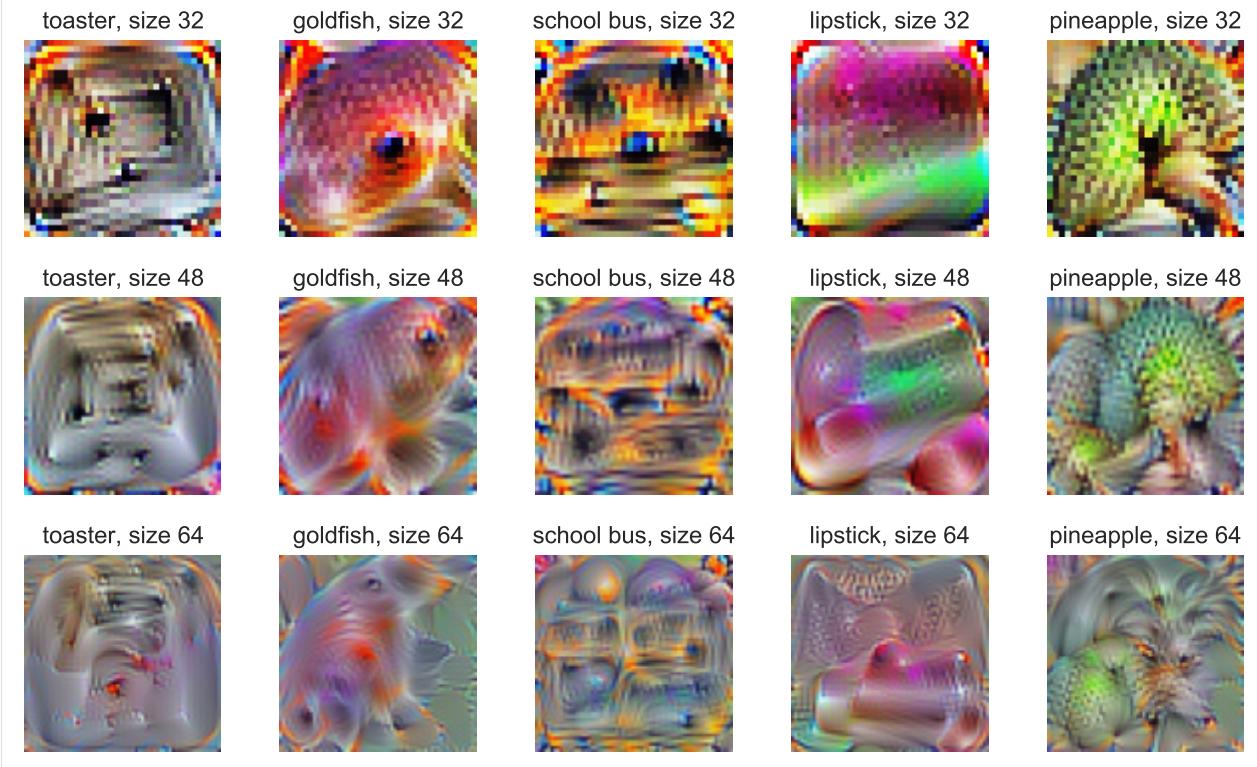
(continues on next page)

(continued from previous page)

```

        ax[p_idx][c_idx].set_title(f"{cname}, size {psize}")
        ax[p_idx][c_idx].axis('off')
fig.subplots_adjust(hspace=0.3, wspace=0.3)
plt.show()
show_patches()

```



We can see a clear difference between patches of different classes and sizes. In the smallest size, 32×32 pixels, some of the patches clearly resemble their class. For instance, the goldfish patch clearly shows a goldfish. The eye and the color are very characteristic of the class. Overall, the patches with 32 pixels have very strong colors that are typical for their class (yellow school bus, pink lipstick, greenish pineapple). The larger the patch becomes, the more stretched the pattern becomes. For the goldfish, we can still spot regions that might represent eyes and the characteristic orange color, but it is not clearly a single fish anymore. For the pineapple, we might interpret the top part of the image as the leaves of pineapple fruit, but it is more abstract than our small patches. Nevertheless, we can easily spot the alignment of the patch to class, even on the largest scale.

Let's now look at the quantitative results.

```
[20]: %%html
<!-- Some HTML code to increase font size in the following table -->
<style>
th {font-size: 120%;}
td {font-size: 120%;}
</style>
<IPython.core.display.HTML object>
```

```
[21]: import tabulate
from IPython.display import display, HTML
```

(continues on next page)

(continued from previous page)

```
def show_table(top_1=True):
    i = 0 if top_1 else 1
    table = [[name] + [f"{(100.0 * patch_dict[name][psize]['results'][i]):.2f}%" for psize in patch_sizes]
              for name in class_names]
    display(HTML(tabulate.tabulate(table, tablefmt='html', headers=["Class name"] + [f
        "Patch size {psize}x{psize}" for psize in patch_sizes])))
```

First, we will create a table of top-1 accuracy, meaning that how many images have been classified with the target class as highest prediction?

[22]: `show_table(top_1=True)`

<IPython.core.display.HTML object>

The clear trend, that we would also have expected, is that the larger the patch, the easier it is to fool the model. For the largest patch size of 64×64 , we are able to fool the model on almost all images, despite the patch covering only 8% of the image. The smallest patch actually covers 2% of the image, which is almost neglectable. Still, the fooling accuracy is quite remarkable. A large variation can be however seen across classes. While *school bus* and *pineapple* seem to be classes that were easily predicted, *toaster* and *lipstick* seem to be much harder for creating a patch. It is hard to intuitively explain why our patches underperform on those classes. Nonetheless, a fooling accuracy of >40% is still very good for such a tiny patch.

Let's also take a look at the top-5 accuracy:

[23]: `show_table(top_1=False)`

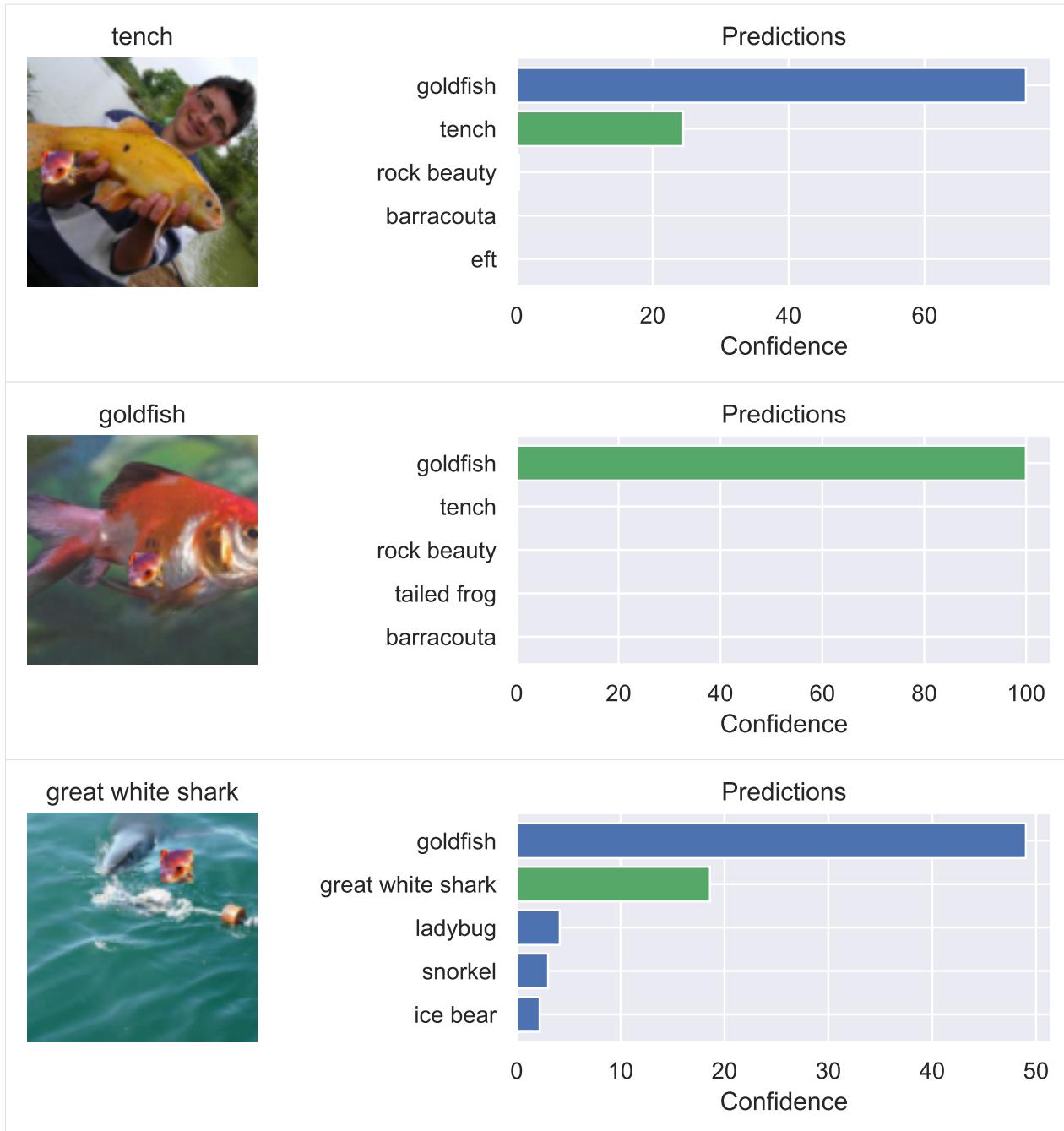
<IPython.core.display.HTML object>

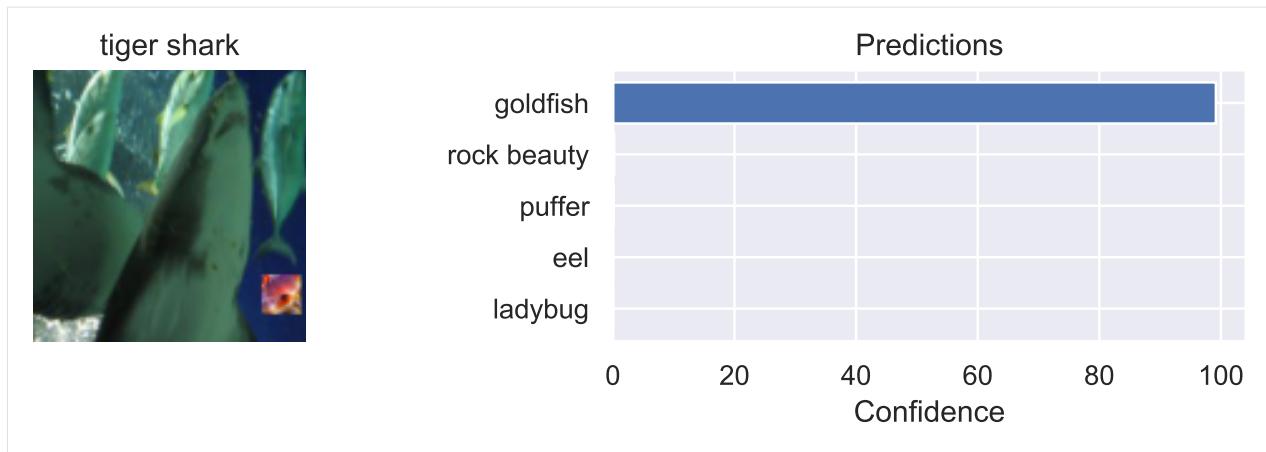
We see a very similar pattern across classes and patch sizes. The patch size 64 obtains >99.7% top-5 accuracy for any class, showing that we can almost fool the network on any image. A top-5 accuracy of >70% for the hard classes and small patches is still impressive and shows how vulnerable deep CNNs are to such attacks.

Finally, let's create some example visualizations of the patch attack in action.

[24]: `def perform_patch_attack(patch):
 patch_batch = exmp_batch.clone()
 patch_batch = place_patch(patch_batch, patch)
 with torch.no_grad():
 patch_preds = pretrained_model(patch_batch.to(device))
 for i in range(1,17,5):
 show_prediction(patch_batch[i], label_batch[i], patch_preds[i])`

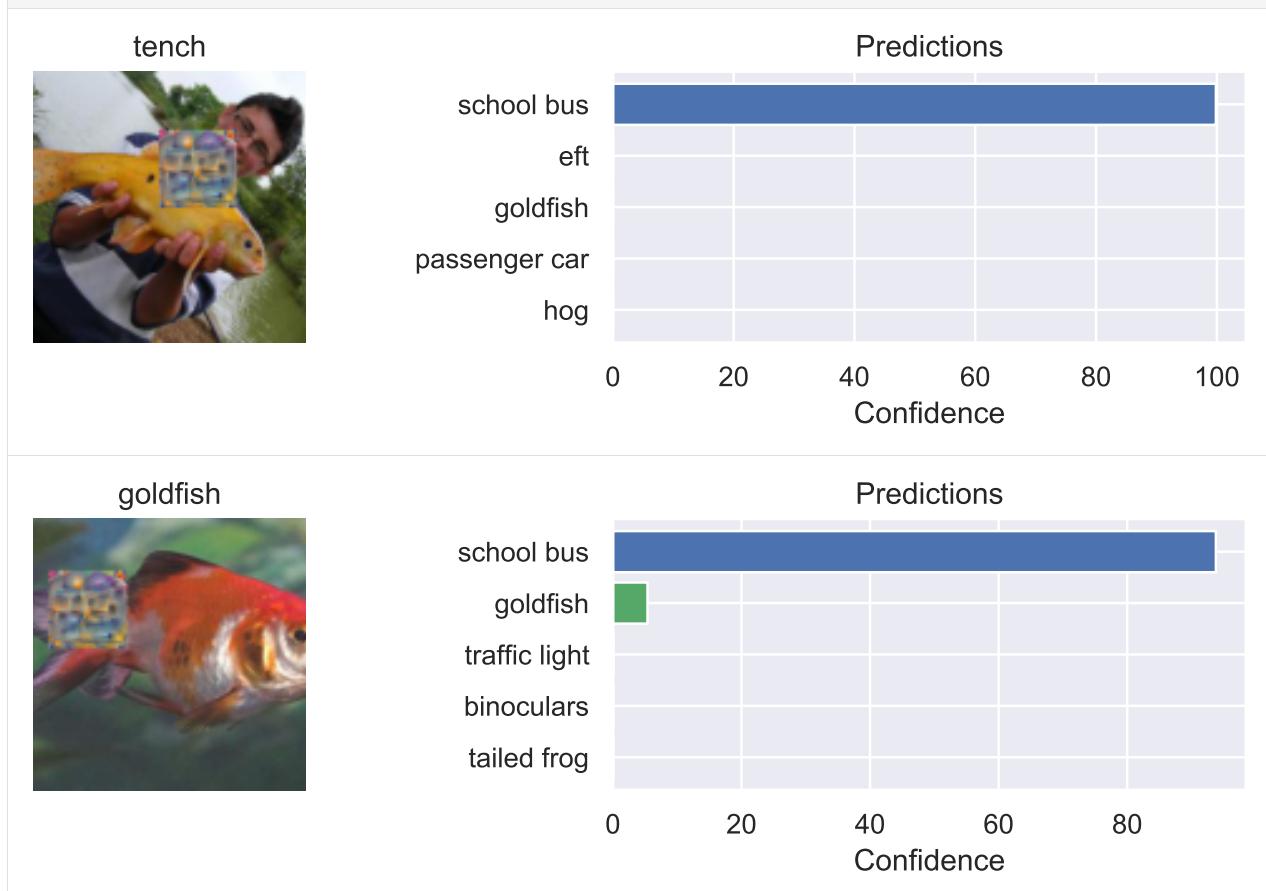
[25]: `perform_patch_attack(patch_dict['goldfish'][32]['patch'])`

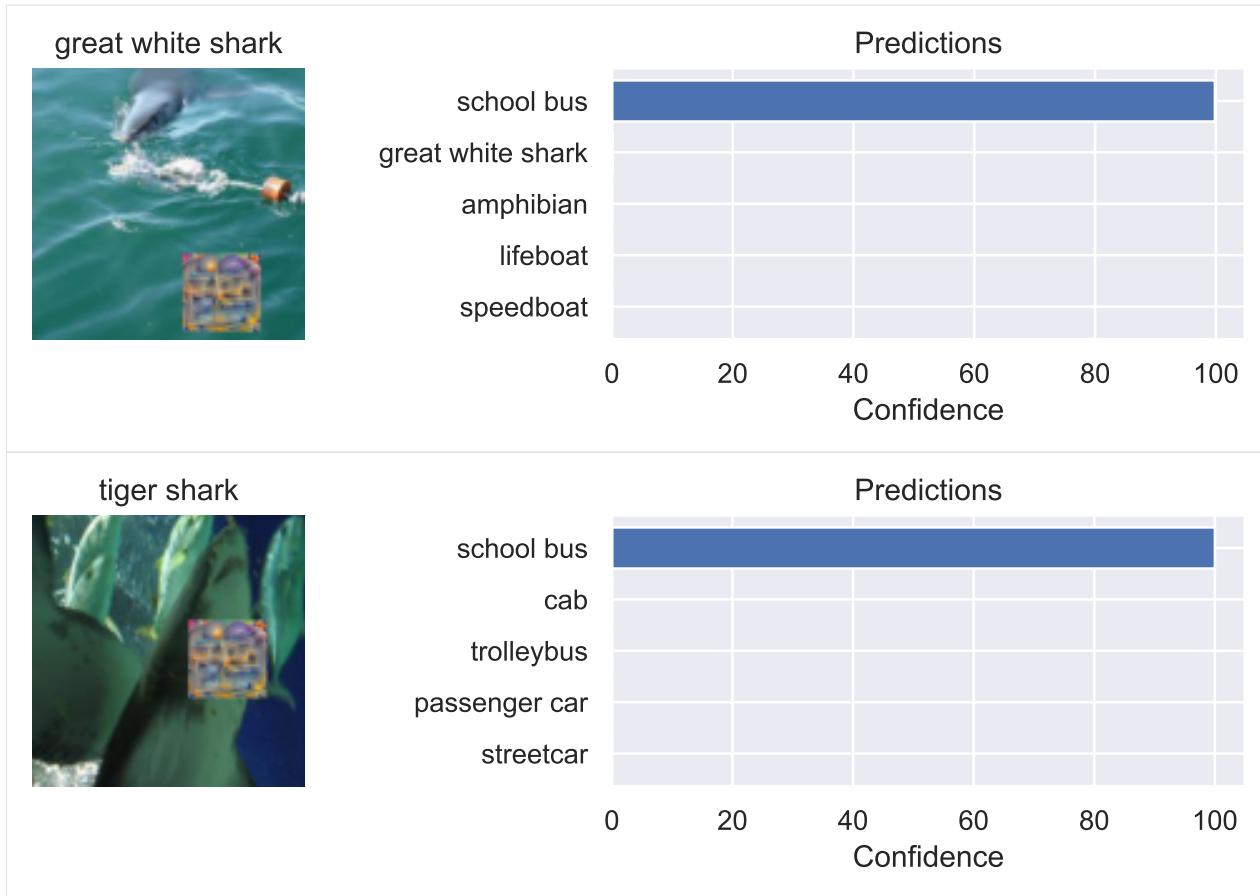




The tiny goldfish patch can change all of the predictions to “goldfish” as top class. Note that the patch attacks work especially well if the input image is semantically similar to the target class (e.g. a fish and the target class “goldfish” works better than an airplane image with that patch). Nevertheless, we can also let the network predict semantically dis-similar classes by using a larger patch:

```
[26]: perform_patch_attack(patch_dict['school bus'][64]['patch'])
```





Although none of the images have anything to do with an American school bus, the high confidence of often 100% shows how powerful such attacks can be. With a few lines of code and access to the model, we were able to generate patches that we add to any image to make the model predict any class we want.

Transferability of white-box attacks

FGSM and the adversarial patch attack were both focused on one specific image. However, can we transfer those attacks to other models? The adversarial patch attack as proposed in [Brown et al.](#), was originally trained on multiple models, and hence was also able to work on many different network architecture. But how different are the patches for different models anyway? For instance, let's evaluate some of our patches trained above on a different network, e.g. DenseNet121.

```
[27]: transfer_model = torchvision.models.densenet121(weights='IMAGENET1K_V1')
transfer_model = transfer_model.to(device)

# No gradients needed for the network
transfer_model.eval()
for p in transfer_model.parameters():
    p.requires_grad = False
```

Feel free to change the class name and/or patch size below to test out different patches.

```
[28]: class_name = 'pineapple'
patch_size = 64
```

(continues on next page)

(continued from previous page)

```

print(f"Testing patch \'{class_name}\' of size {patch_size}x{patch_size}")

results = eval_patch(transfer_model,
                     patch_dict[class_name][patch_size][“patch”],
                     data_loader,
                     target_class=label_names.index(class_name))

print(f"Top-1 fool accuracy: {(results[0] * 100.0):.2f}%")
print(f"Top-5 fool accuracy: {(results[1] * 100.0):.2f}%")

Testing patch "pineapple" of size 64x64
HBox(children=(FloatProgress(value=0.0, description='Validating...', max=157.0, ↴
style=ProgressStyle(descriptio...
Top-1 fool accuracy: 65.28%
Top-5 fool accuracy: 82.58%

```

Although the fool accuracy is significantly lower than on the original ResNet34, it still has a considerable impact on DenseNet although the networks have completely different architectures and weights. If you would compare more patches and models, some would work better than others. However, one aspect which allows patch attacks to generalize well is if all the networks have been trained on the same data. In this case, all networks have been trained on ImageNet. Dataset biases make the networks recognize specific patterns in the underlying image data that humans would not have seen, and/or only work for the given dataset. This is why the knowledge of what data has been used to train a specific model is already worth a lot in the context of adversarial attacks.

4.13.3 Protecting against adversarial attacks

There are many more attack strategies than just FGSM and adversarial patches that we haven't discussed and implemented ourselves here. However, what about the other perspective? What can we do to *protect* a network against adversarial attacks? The sad truth to this is: not much.

White-box attacks require access to the model and its gradient calculation. The easiest way of preventing this is by ensuring safe, private storage of the model and its weights. However, some attacks, called black-box attacks, also work without access to the model's parameters, or white-box attacks can also generalize as we have seen above on our short test on transferability.

So, how could we eventually protect a model? An intuitive approach would to train/fine-tune a model on such adversarial images, leading to an adversarial training similar to a GAN. During training, we would pretend to be the attacker, and use for example FGSM as an augmentation strategy. However, this usually just ends up in an oscillation of the defending network between weak spots. Another common trick to increase robustness against adversarial attacks is defensive distillation (Papernot et al.). Instead of training the model on the dataset labels, we train a secondary model on the softmax predictions of the first one. This way, the loss surface is “smoothed” in the directions an attacker might try to exploit, and it becomes more difficult for the attacker to find adversarial examples. Nevertheless, there hasn't been found the one, true strategy that works against all possible adversarial attacks.

Why are CNNs, or neural networks in general, so vulnerable to adversarial attacks? While there are many possible explanations, the most intuitive is that neural networks don't know what they don't know. Even a large dataset represents just a few sparse points in the extremely large space of possible images. A lot of the input space has not been seen by the network during training, and hence, we cannot guarantee that the prediction for those images is any useful. The network instead learns a very good classification on a smaller region, often referred to as manifold, while ignoring the points outside of it. NNs with uncertainty prediction could potentially help to discover what the network does not know. Another possible explanation lies in the activation function. As we know, most CNNs use ReLU-based activation functions. While those have enabled great success in training deep neural networks due to their stable gradient for positive values, they also constitute a possible flaw. The output range of a ReLU neuron can be arbitrarily high. Thus,

if we design a patch or the noise in the image to cause a very high value for a single neuron, it can overpower many other features in the network. Thus, although ReLU stabilizes training, it also offers a potential point of attack for adversaries.

4.13.4 Conclusion

In this tutorial, we have looked at different forms of adversarial attacks. Deep CNNs can be fooled by only slight modifications to the input. Whether it is a carefully designed noise pattern, unnoticeable for a human, or a small patch, we are able to manipulate the networks' predictions significantly. The fact that even white-box attacks can be transferable across networks, and that there exist no suitable protections against all possible adversarial attacks, make this concept a massive problem for real-world applications. While adversarial attacks can also be used for improving/training a robust model or a GAN, it is not close to being solved yet. This is also because neural networks are currently complex, unknown non-linear functions in high-dimensional looking for correlations instead of causation. In the next years, we might hopefully see an improvement in the stability of such models by using causal approaches and/or introducing uncertainty.

4.13.5 References

- [1] Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples." ICLR 2015.
 - [2] Hendrik Metzen, Jan, et al. "Universal adversarial perturbations against semantic image segmentation." Proceedings of the IEEE International Conference on Computer Vision. 2017.
 - [3] Anant Jain. "Breaking neural networks with adversarial attacks." [Blog post](#) 2019.
-

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.14 Tutorial 11: Normalizing Flows for image modeling

Filled notebook:

Pre-trained models:

Recordings:

JAX+Flax version:

Author: Phillip Lippe

Note: Interested in JAX? Check out our [JAX+Flax version](#) of this tutorial!

In this tutorial, we will take a closer look at complex, deep normalizing flows. The most popular, current application of deep normalizing flows is to model datasets of images. As for other generative models, images are a good domain to start working on because (1) CNNs are widely studied and strong models exist, (2) images are high-dimensional and complex, and (3) images are discrete integers. In this tutorial, we will review current advances in normalizing flows for image modeling, and get hands-on experience on coding normalizing flows. Note that normalizing flows are commonly parameter heavy and therefore computationally expensive. We will use relatively simple and shallow flows to save computational cost and allow you to run the notebook on CPU, but keep in mind that a simple way to improve the scores of the flows we study here is to make them deeper.

Throughout this notebook, we make use of PyTorch Lightning. The first cell imports our usual libraries.

```
[1]: ## Standard libraries
import os
import math
import time
import numpy as np

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## Progress bar
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
# Torchvision
import torchvision
from torchvision.datasets import MNIST
from torchvision import transforms
# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial11"
```

(continues on next page)

(continued from previous page)

```
# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Fetching the device that will be used throughout this notebook
device = torch.device("cpu") if not torch.cuda.is_available() else torch.device("cuda:0")
print("Using device", device)

Global seed set to 42
Using device cuda:0
```

Again, we have a few pretrained models. We download them below to the specified path above.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial11/"
# Files to download
pretrained_files = ["MNISTFlow_simple.ckpt", "MNISTFlow_vardeq.ckpt", "MNISTFlow_"
    ↴multiscale.ckpt"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive_"
                ↴folder, or contact the author with the full output including the following error:\n", ↴
                ↴e)
```

We will use the MNIST dataset in this notebook. MNIST constitutes, despite its simplicity, a challenge for small generative models as it requires the global understanding of an image. At the same time, we can easily judge whether generated images come from the same distribution as the dataset (i.e. represent real digits), or not.

To deal better with the discrete nature of the images, we transform them from a range of 0-1 to a range of 0-255 as integers.

```
[3]: # Convert images from 0-1 to 0-255 (integers)
def discretize(sample):
    return (sample * 255).to(torch.int32)

# Transformations applied on each image => make them a tensor and discretize
transform = transforms.Compose([transforms.ToTensor(),
```

(continues on next page)

(continued from previous page)

```

discretize])

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = MNIST(root=DATASET_PATH, train=True, transform=transform, download=True)
pl.seed_everything(42)
train_set, val_set = torch.utils.data.random_split(train_dataset, [50000, 10000])

# Loading the test set
test_set = MNIST(root=DATASET_PATH, train=False, transform=transform, download=True)

# We define a set of data loaders that we can use for various purposes later.
# Note that for actually training a model, we will use different data loaders
# with a lower batch size.
train_loader = data.DataLoader(train_set, batch_size=256, shuffle=False, drop_last=False)
val_loader = data.DataLoader(val_set, batch_size=64, shuffle=False, drop_last=False, num_
    ↪workers=4)
test_loader = data.DataLoader(test_set, batch_size=64, shuffle=False, drop_last=False, ↪
    ↪num_workers=4)

Global seed set to 42

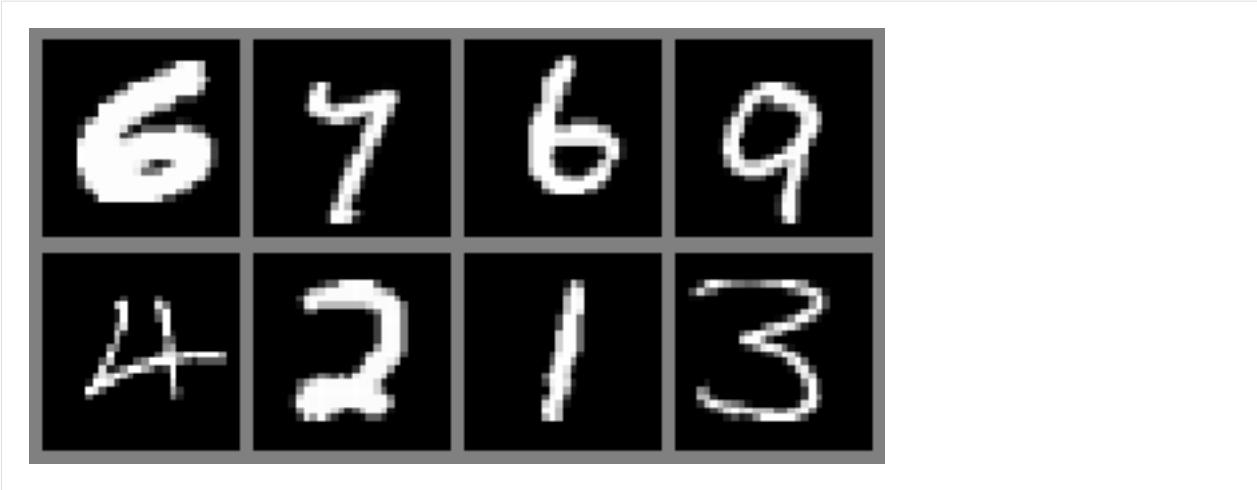
```

In addition, we will define below a function to simplify the visualization of images/samples. Some training examples of the MNIST dataset is shown below.

```
[4]: def show_imgs(imgs, title=None, row_size=4):
    # Form a grid of pictures (we use max. 8 columns)
    num_imgs = imgs.shape[0] if isinstance(imgs, torch.Tensor) else len(imgs)
    is_int = imgs.dtype==torch.int32 if isinstance(imgs, torch.Tensor) else imgs[0].
        ↪dtype==torch.int32
    nrow = min(num_imgs, row_size)
    ncol = int(math.ceil(num_imgs/nrow))
    imgs = torchvision.utils.make_grid(imgs, nrow=nrow, pad_value=128 if is_int else 0.5)
    np_imgs = imgs.cpu().numpy()
    # Plot the grid
    plt.figure(figsize=(1.5*nrow, 1.5*ncol))
    plt.imshow(np.transpose(np_imgs, (1,2,0)), interpolation='nearest')
    plt.axis('off')
    if title is not None:
        plt.title(title)
    plt.show()
    plt.close()

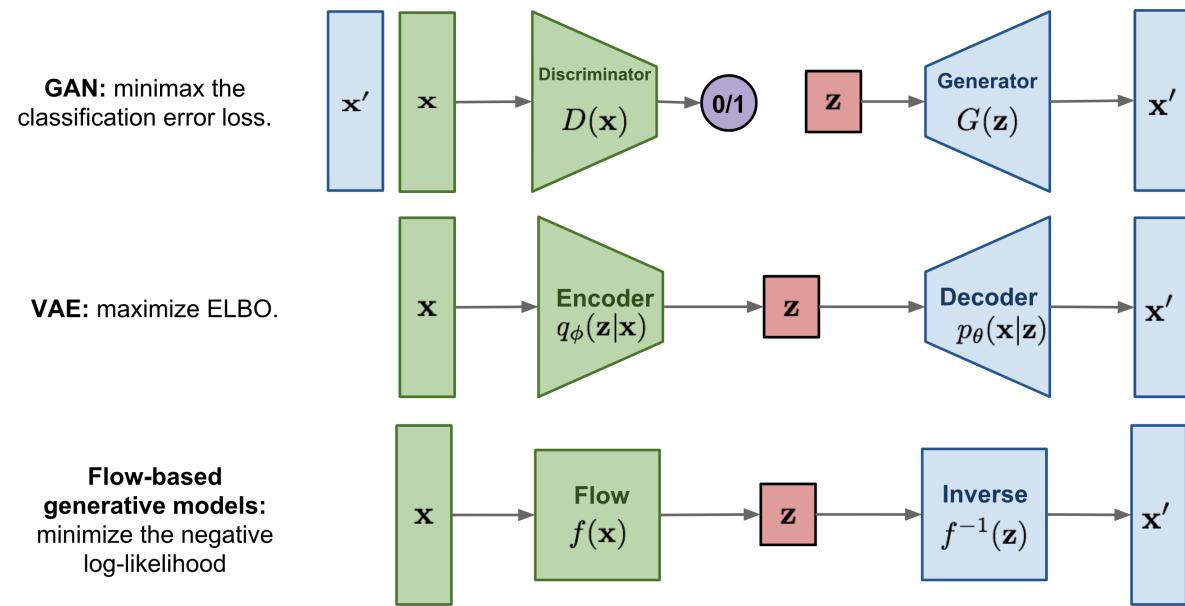
show_imgs([train_set[i][0] for i in range(8)])

```



4.14.1 Normalizing Flows as generative model

In the previous lectures, we have seen Energy-based models, Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs) as example of generative models. However, none of them explicitly learn the probability density function $p(x)$ of the real input data. While VAEs model a lower bound, energy-based models only implicitly learn the probability density. GANs on the other hand provide us a sampling mechanism for generating new data, without offering a likelihood estimate. The generative model we will look at here, called Normalizing Flows, actually models the true data distribution $p(x)$ and provides us with an exact likelihood estimate. Below, we can visually compare VAEs, GANs and Flows (figure credit - [Lilian Weng](#)):



The major difference compared to VAEs is that flows use *invertible* functions f to map the input data x to a latent representation z . To realize this, z must be of the same shape as x . This is in contrast to VAEs where z is usually much lower dimensional than the original input data. However, an invertible mapping also means that for every data point x , we have a corresponding latent representation z which allows us to perform lossless reconstruction (z to x). In the visualization above, this means that $x = x'$ for flows, no matter what invertible function f and input x we choose.

Nonetheless, how are normalizing flows modeling a probability density with an invertible function? The answer to this question is the rule for change of variables. Specifically, given a prior density $p_z(z)$ (e.g. Gaussian) and an invertible function f , we can determine $p_x(x)$ as follows:

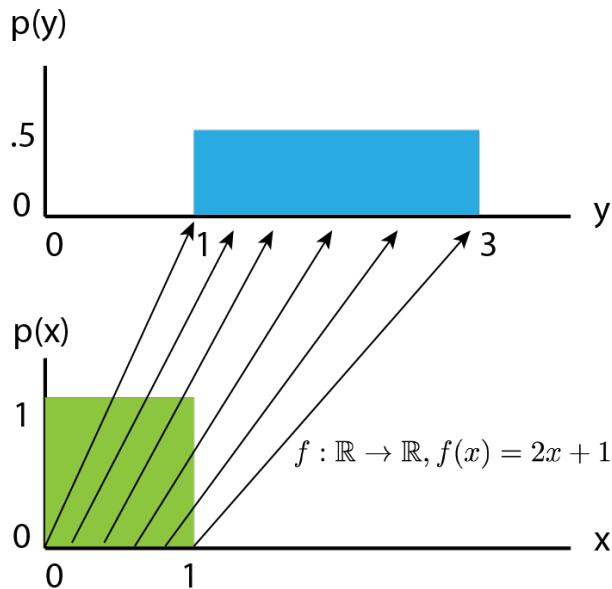
$$\int p_x(x)dx = \int p_z(z)dz = 1 \quad (\text{by definition of a probability distribution})$$

$$\Leftrightarrow p_x(x) = p_z(z) \left| \frac{dz}{dx} \right| = p_z(f(x)) \left| \frac{df(x)}{dx} \right|$$

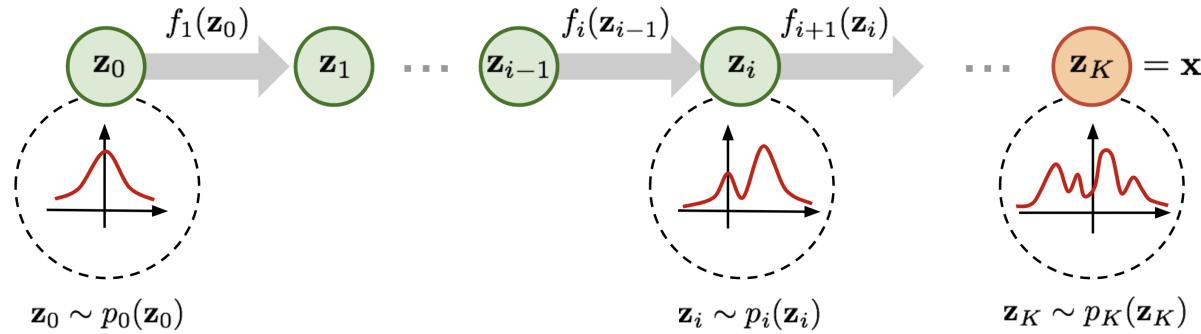
Hence, in order to determine the probability of x , we only need to determine its probability in latent space, and get the derivative of f . Note that this is for a univariate distribution, and f is required to be invertible and smooth. For a multivariate case, the derivative becomes a Jacobian of which we need to take the determinant. As we usually use the log-likelihood as objective, we write the multivariate term with logarithms below:

$$\log p_x(\mathbf{x}) = \log p_z(f(\mathbf{x})) + \log \left| \det \frac{df(\mathbf{x})}{d\mathbf{x}} \right|$$

Although we now know how a normalizing flow obtains its likelihood, it might not be clear what a normalizing flow does intuitively. For this, we should look from the inverse perspective of the flow starting with the prior probability density $p_z(z)$. If we apply an invertible function on it, we effectively “transform” its probability density. For instance, if $f^{-1}(z) = z + 1$, we shift the density by one while still remaining a valid probability distribution, and being invertible. We can also apply more complex transformations, like scaling: $f^{-1}(z) = 2z + 1$, but there you might see a difference. When you scale, you also change the volume of the probability density, as for example on uniform distributions (figure credit - [Eric Jang](#)):



You can see that the height of $p(y)$ should be lower than $p(x)$ after scaling. This change in volume represents $\left| \frac{df(x)}{dx} \right|$ in our equation above, and ensures that even after scaling, we still have a valid probability distribution. We can go on with making our function f more complex. However, the more complex f becomes, the harder it will be to find the inverse f^{-1} of it, and to calculate the log-determinant of the Jacobian $\log \left| \det \frac{df(\mathbf{x})}{d\mathbf{x}} \right|$. An easier trick is to stack multiple invertible functions f_1, \dots, f_K after each other, as all together, they still represent a single, invertible function. Using multiple, learnable invertible functions, a normalizing flow attempts to transform $p_z(z)$ slowly into a more complex distribution which should finally be $p_x(x)$. We visualize the idea below (figure credit - [Lilian Weng](#)):



Starting from z_0 , which follows the prior Gaussian distribution, we sequentially apply the invertible functions f_1, f_2, \dots, f_K , until z_K represents x . Note that in the figure above, the functions f represent the inverted function from f we had above (here: $f : Z \rightarrow X$, above: $f : X \rightarrow Z$). This is just a different notation and has no impact on the actual flow design because all f need to be invertible anyways. When we estimate the log likelihood of a data point x as in the equations above, we run the flows in the opposite direction than visualized above. Multiple flow layers have been proposed that use a neural network as learnable parameters, such as the planar and radial flow. However, we will focus here on flows that are commonly used in image modeling, and will discuss them in the rest of the notebook along with the details of how to train a normalizing flow.

4.14.2 Normalizing Flows on images

To become familiar with normalizing flows, especially for the application of image modeling, it is best to discuss the different elements in a flow along with the implementation. As a general concept, we want to build a normalizing flow that maps an input image (here MNIST) to an equally sized latent space:

As a first step, we will implement a template of a normalizing flow in PyTorch Lightning. During training and validation, a normalizing flow performs density estimation in the forward direction. For this, we apply a series of flow transformations on the input x and estimate the probability of the input by determining the probability of the transformed point z given a prior, and the change of volume caused by the transformations. During inference, we can do both density estimation and sampling new points by inverting the flow transformations. Therefore, we define a function `_get_likelihood` which performs density estimation, and `sample` to generate new examples. The functions `training_step`, `validation_step` and `test_step` all make use of `_get_likelihood`.

The standard metric used in generative models, and in particular normalizing flows, is bits per dimensions (bpd). Bpd is motivated from an information theory perspective and describes how many bits we would need to encode a particular example in our modeled distribution. The less bits we need, the more likely the example in our distribution. When we test for the bits per dimension of our test dataset, we can judge whether our model generalizes to new samples of the dataset and didn't memorize the training dataset. In order to calculate the bits per dimension score, we can rely on the negative log-likelihood and change the log base (as bits are binary while NLL is usually exponential):

$$\text{bpd} = \text{nll} \cdot \log_2(\exp(1)) \cdot \left(\prod d_i \right)^{-1}$$

where d_1, \dots, d_K are the dimensions of the input. For images, this would be the height, width and channel number. We divide the log likelihood by these extra dimensions to have a metric which we can compare for different image resolutions. In the original image space, MNIST examples have a bits per dimension score of 8 (we need 8 bits to encode each pixel as there are 256 possible values).

```
[5]: class ImageFlow(pl.LightningModule):
```

(continues on next page)

(continued from previous page)

```

def __init__(self, flows, import_samples=8):
    """
    Inputs:
        flows - A list of flows (each a nn.Module) that should be applied on the
    ↪images.
        import_samples - Number of importance samples to use during testing (see
    ↪explanation below). Can be changed at any time
    """
    super().__init__()
    self.flows = nn.ModuleList(flows)
    self.import_samples = import_samples
    # Create prior distribution for final latent space
    self.prior = torch.distributions.normal.Normal(loc=0.0, scale=1.0)
    # Example input for visualizing the graph
    self.example_input_array = train_set[0][0].unsqueeze(dim=0)

def forward(self, imgs):
    # The forward function is only used for visualizing the graph
    return self._get_likelihood(imgs)

def encode(self, imgs):
    # Given a batch of images, return the latent representation z and ldj of the
    ↪transformations
    z, ldj = imgs, torch.zeros(imgs.shape[0], device=self.device)
    for flow in self.flows:
        z, ldj = flow(z, ldj, reverse=False)
    return z, ldj

def _get_likelihood(self, imgs, return_ll=False):
    """
    Given a batch of images, return the likelihood of those.
    If return_ll is True, this function returns the log likelihood of the input.
    Otherwise, the output metric is bits per dimension (scaled negative log-
    ↪likelihood)
    """
    z, ldj = self.encode(imgs)
    log_pz = self.prior.log_prob(z).sum(dim=[1,2,3])
    log_px = ldj + log_pz
    nll = -log_px
    # Calculating bits per dimension
    bpd = nll * np.log2(np.exp(1)) / np.prod(imgs.shape[1:])
    return bpd.mean() if not return_ll else log_px

@torch.no_grad()
def sample(self, img_shape, z_init=None):
    """
    Sample a batch of images from the flow.
    """
    # Sample latent representation from prior
    if z_init is None:
        z = self.prior.sample(sample_shape=img_shape).to(device)
    else:

```

(continues on next page)

(continued from previous page)

```

z = z_init.to(device)

# Transform z to x by inverting the flows
ldj = torch.zeros(img_shape[0], device=device)
for flow in reversed(self.flows):
    z, ldj = flow(z, ldj, reverse=True)
return z

def configure_optimizers(self):
    optimizer = optim.Adam(self.parameters(), lr=1e-3)
    # An scheduler is optional, but can help in flows to get the last bpd improvement
    scheduler = optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.99)
    return [optimizer], [scheduler]

def training_step(self, batch, batch_idx):
    # Normalizing flows are trained by maximum likelihood => return bpd
    loss = self._get_likelihood(batch[0])
    self.log('train_bpd', loss)
    return loss

def validation_step(self, batch, batch_idx):
    loss = self._get_likelihood(batch[0])
    self.log('val_bpd', loss)

def test_step(self, batch, batch_idx):
    # Perform importance sampling during testing => estimate likelihood M times for
    each image
    samples = []
    for _ in range(self.import_samples):
        img_ll = self._get_likelihood(batch[0], return_ll=True)
        samples.append(img_ll)
    img_ll = torch.stack(samples, dim=-1)

    # To average the probabilities, we need to go from log-space to exp, and back to
    log.
    # Logsumexp provides us a stable implementation for this
    img_ll = torch.logsumexp(img_ll, dim=-1) - np.log(self.import_samples)

    # Calculate final bpd
    bpd = -img_ll * np.log2(np.exp(1)) / np.prod(batch[0].shape[1:])
    bpd = bpd.mean()

    self.log('test_bpd', bpd)

```

The `test_step` function differs from the training and validation step in that it makes use of importance sampling. We will discuss the motivation and details behind this after understanding how flows model discrete images in continuous space.

Dequantization

Normalizing flows rely on the rule of change of variables, which is naturally defined in continuous space. Applying flows directly on discrete data leads to undesired density models where arbitrarily high likelihood are placed on a few, particular values. See the illustration below:

The black points represent the discrete points, and the green volume the density modeled by a normalizing flow in continuous space. The flow would continue to increase the likelihood for $x = 0, 1, 2, 3$ while having no volume on any other point. Remember that in continuous space, we have the constraint that the overall volume of the probability density must be 1 ($\int p(x)dx = 1$). Otherwise, we don't model a probability distribution anymore. However, the discrete points $x = 0, 1, 2, 3$ represent delta peaks with no width in continuous space. This is why the flow can place an infinite high likelihood on these few points while still representing a distribution in continuous space. Nonetheless, the learned density does not tell us anything about the distribution among the discrete points, as in discrete space, the likelihoods of those four points would have to sum to 1, not to infinity.

To prevent such degenerated solutions, a common solution is to add a small amount of noise to each discrete value, which is also referred to as dequantization. Considering x as an integer (as it is the case for images), the dequantized representation v can be formulated as $v = x + u$ where $u \in [0, 1]^D$. Thus, the discrete value 1 is modeled by a distribution over the interval [1.0, 2.0], the value 2 by an volume over [2.0, 3.0], etc. Our objective of modeling $p(x)$ becomes:

$$p(x) = \int p(x+u)du = \int \frac{q(u|x)}{q(u|x)} p(x+u)du = \mathbb{E}_{u \sim q(u|x)} \left[\frac{p(x+u)}{q(u|x)} \right]$$

with $q(u|x)$ being the noise distribution. For now, we assume it to be uniform, which can also be written as $p(x) = \mathbb{E}_{u \sim U(0,1)^D} [p(x+u)]$.

In the following, we will implement Dequantization as a flow transformation itself. After adding noise to the discrete values, we additionally transform the volume into a Gaussian-like shape. This is done by scaling $x+u$ between 0 and 1, and applying the invert of the sigmoid function $\sigma(z)^{-1} = \log z - \log 1 - z$. If we would not do this, we would face two problems:

1. The input is scaled between 0 and 256 while the prior distribution is a Gaussian with mean 0 and standard deviation 1. In the first iterations after initializing the parameters of the flow, we would have extremely low likelihoods for large values like 256. This would cause the training to diverge instantaneously.
2. As the output distribution is a Gaussian, it is beneficial for the flow to have a similarly shaped input distribution. This will reduce the modeling complexity that is required by the flow.

Overall, we can implement dequantization as follows:

```
[6]: class Dequantization(nn.Module):
    def __init__(self, alpha=1e-5, quants=256):
        """
        Inputs:
            alpha - small constant that is used to scale the original input.
                    Prevents dealing with values very close to 0 and 1 when inverting
                    the sigmoid
            quants - Number of possible discrete values (usually 256 for 8-bit image)
        """
        super().__init__()
        self.alpha = alpha
        self.quants = quants

    def forward(self, z, ldj, reverse=False):
```

(continues on next page)

(continued from previous page)

```

if not reverse:
    z, ldj = self.dequant(z, ldj)
    z, ldj = self.sigmoid(z, ldj, reverse=True)
else:
    z, ldj = self.sigmoid(z, ldj, reverse=False)
    z = z * self.quants
    ldj += np.log(self.quants) * np.prod(z.shape[1:])
    z = torch.floor(z).clamp(min=0, max=self.quants-1).to(torch.int32)
return z, ldj

def sigmoid(self, z, ldj, reverse=False):
    # Applies an invertible sigmoid transformation
    if not reverse:
        ldj += (-z-2*F.softplus(-z)).sum(dim=[1,2,3])
        z = torch.sigmoid(z)
        # Reversing scaling for numerical stability
        ldj -= np.log(1 - self.alpha) * np.prod(z.shape[1:])
        z = (z - 0.5 * self.alpha) / (1 - self.alpha)
    else:
        z = z * (1 - self.alpha) + 0.5 * self.alpha # Scale to prevent boundaries 0
        ↵ and 1
        ldj += np.log(1 - self.alpha) * np.prod(z.shape[1:])
        ldj += (-torch.log(z) - torch.log(1-z)).sum(dim=[1,2,3])
        z = torch.log(z) - torch.log(1-z)
    return z, ldj

def dequant(self, z, ldj):
    # Transform discrete values to continuous volumes
    z = z.to(torch.float32)
    z = z + torch.rand_like(z).detach()
    z = z / self.quants
    ldj -= np.log(self.quants) * np.prod(z.shape[1:])
    return z, ldj

```

A good check whether a flow is correctly implemented or not, is to verify that it is invertible. Hence, we will dequantize a randomly chosen training image, and then quantize it again. We would expect that we would get the exact same image out:

```
[7]: ## Testing invertibility of dequantization layer
pl.seed_everything(42)
orig_img = train_set[0][0].unsqueeze(dim=0)
ldj = torch.zeros(1,)
dequant_module = Dequantization()
deq_img, ldj = dequant_module(orig_img, ldj, reverse=False)
reconst_img, ldj = dequant_module(deq_img, ldj, reverse=True)

d1, d2 = torch.where(orig_img.squeeze() != reconst_img.squeeze())
if len(d1) != 0:
    print("Dequantization was not invertible.")
    for i in range(d1.shape[0]):
        print("Original value:", orig_img[0,0,d1[i], d2[i]].item())
        print("Reconstructed value:", reconst_img[0,0,d1[i], d2[i]].item())
```

(continues on next page)

(continued from previous page)

```

else:
    print("Successfully inverted dequantization")

# Layer is not strictly invertible due to float precision constraints
# assert (orig_img == reconst_img).all().item()

Global seed set to 42
Successfully inverted dequantization

```

The test succeeds as we would expect. However, there is a chance that the test fails due to numerical inaccuracies in the sigmoid invert. While the input space to the inverted sigmoid is scaled between 0 and 1, the output space is between $-\infty$ and ∞ . And as we use 32 bits to represent the numbers (in addition to applying logs over and over again), such inaccuracies can occur and should not be worrisome. Nevertheless, it is good to be aware of them, and can be improved by using a double tensor (float64).

Finally, we can take our dequantization and actually visualize the distribution it transforms the discrete values into:

```

[8]: def visualize_dequantization(quants, prior=None):
    """
    Function for visualizing the dequantization values of discrete values in continuous
    space
    """

    # Prior over discrete values. If not given, a uniform is assumed
    if prior is None:
        prior = np.ones(quants, dtype=np.float32) / quants
        prior = prior / prior.sum() # Ensure proper categorical distribution

    inp = torch.arange(-4, 4, 0.01).view(-1, 1, 1, 1) # Possible continuous values we
    ↪want to consider
    ldj = torch.zeros(inp.shape[0])
    dequant_module = Dequantization(quants=quants)
    # Invert dequantization on continuous values to find corresponding discrete value
    out, ldj = dequant_module.forward(inp, ldj, reverse=True)
    inp, out, prob = inp.squeeze().numpy(), out.squeeze().numpy(), ldj.exp().numpy()
    prob = prob * prior[out] # Probability scaled by categorical prior

    # Plot volumes and continuous distribution
    sns.set_style("white")
    fig = plt.figure(figsize=(6,3))
    x_ticks = []
    for v in np.unique(out):
        indices = np.where(out==v)
        color = to_rgb(f"C{v}")
        plt.fill_between(inp[indices], prob[indices], np.zeros(indices[0].shape[0]), ↪
        ↪color=color+(0.5,), label=str(v))
        plt.plot([inp[indices[0][0]]]*2, [0, prob[indices[0][0]]], color=color)
        plt.plot([inp[indices[0][-1]]]*2, [0, prob[indices[0][-1]]], color=color)
        x_ticks.append(inp[indices[0][0]])
    x_ticks.append(inp.max())
    plt.xticks(x_ticks, [f'{x:.1f}' for x in x_ticks])
    plt.plot(inp, prob, color=(0.0, 0.0, 0.0))
    # Set final plot properties
    plt.ylim(0, prob.max()*1.1)

```

(continues on next page)

(continued from previous page)

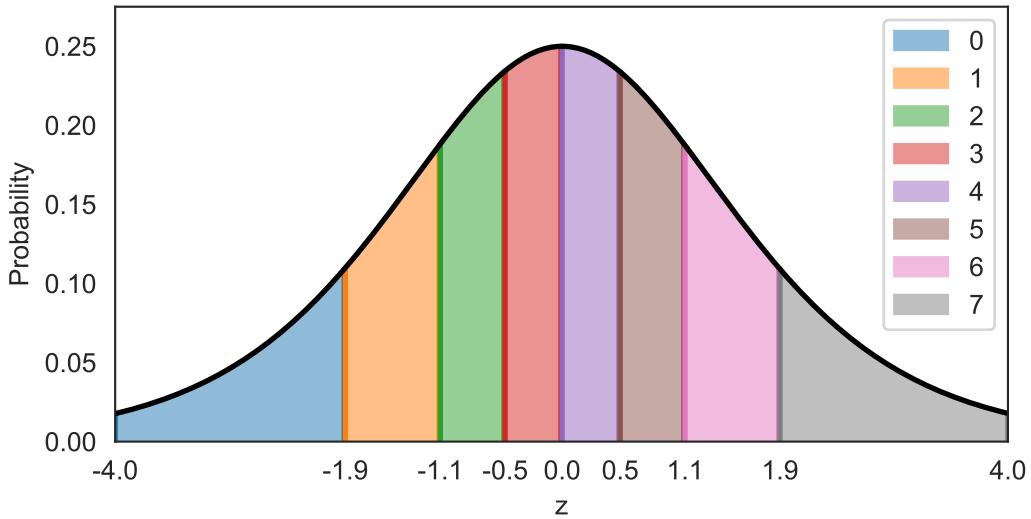
```

plt.xlim(inp.min(), inp.max())
plt.xlabel("z")
plt.ylabel("Probability")
plt.title(f"Dequantization distribution for {quants} discrete values")
plt.legend()
plt.show()
plt.close()

visualize_dequantization(quants=8)

```

Dequantization distribution for 8 discrete values



The visualized distribution show the sub-volumes that are assigned to the different discrete values. The value 0 has its volume between $[-\infty, -1.9]$, the value 1 is represented by the interval $[-1.9, -1.1]$, etc. The volume for each discrete value has the same probability mass. That's why the volumes close to the center (e.g. 3 and 4) have a smaller area on the z-axis as others (z is being used to denote the output of the whole dequantization flow).

Effectively, the consecutive normalizing flow models discrete images by the following objective:

$$\log p(x) = \log \mathbb{E}_{u \sim q(u|x)} \left[\frac{p(x+u)}{q(u|x)} \right] \geq \mathbb{E}_u \left[\log \frac{p(x+u)}{q(u|x)} \right]$$

Although normalizing flows are exact in likelihood, we have a lower bound. Specifically, this is an example of the Jensen inequality because we need to move the log into the expectation so we can use Monte-carlo estimates. In general, this bound is considerably smaller than the ELBO in variational autoencoders. Actually, we can reduce the bound ourselves by estimating the expectation not by one, but by M samples. In other words, we can apply importance sampling which leads to the following inequality:

$$\log p(x) = \log \mathbb{E}_{u \sim q(u|x)} \left[\frac{p(x+u)}{q(u|x)} \right] \geq \mathbb{E}_u \left[\log \frac{1}{M} \sum_{m=1}^M \frac{p(x+u_m)}{q(u_m|x)} \right] \geq \mathbb{E}_u \left[\log \frac{p(x+u)}{q(u|x)} \right]$$

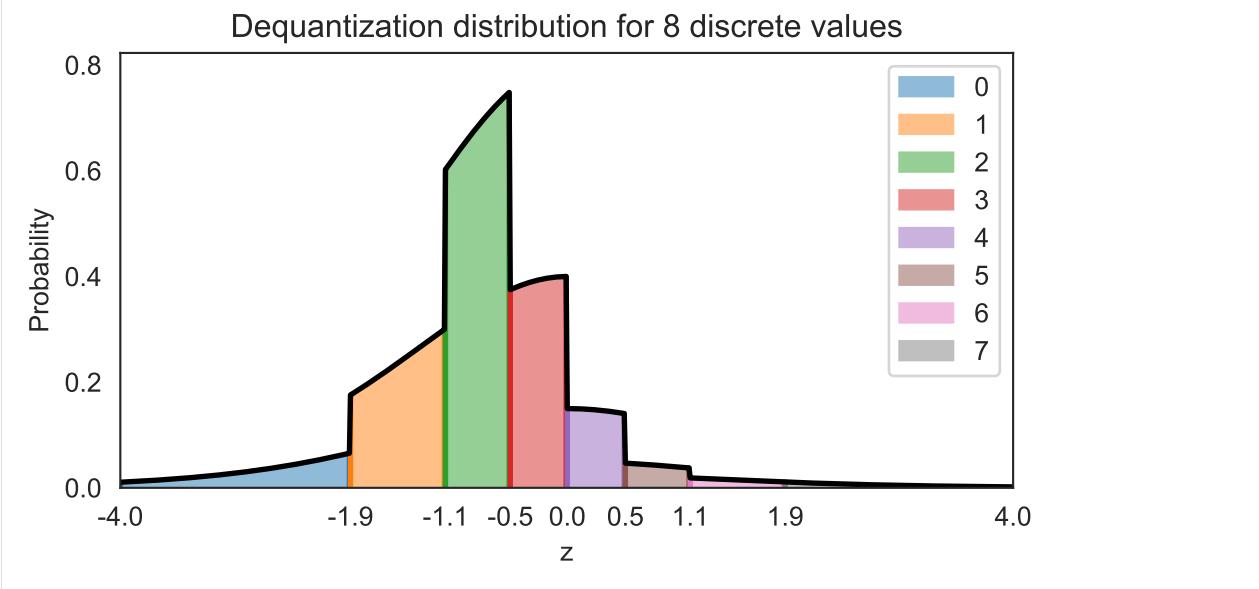
The importance sampling $\frac{1}{M} \sum_{m=1}^M \frac{p(x+u_m)}{q(u_m|x)}$ becomes $\mathbb{E}_{u \sim q(u|x)} \left[\frac{p(x+u)}{q(u|x)} \right]$ if $M \rightarrow \infty$, so that the more samples we use, the tighter the bound is. During testing, we can make use of this property and have it implemented in `test_step` in `ImageFlow`. In theory, we could also use this tighter bound during training. However, related work has shown that this does not necessarily lead to an improvement given the additional computational cost, and it is more efficient to stick with a single estimate [5].

Variational Dequantization

Dequantization uses a uniform distribution for the noise u which effectively leads to images being represented as hypercubes (cube in high dimensions) with sharp borders. However, modeling such sharp borders is not easy for a flow as it uses smooth transformations to convert it into a Gaussian distribution.

Another way of looking at it is if we change the prior distribution in the previous visualization. Imagine we have independent Gaussian noise on pixels which is commonly the case for any real-world taken picture. Therefore, the flow would have to model a distribution as above, but with the individual volumes scaled as follows:

```
[9]: visualize_dequantization(quants=8, prior=np.array([0.075, 0.2, 0.4, 0.2, 0.075, 0.025, 0.0125, 0.0125]))
```



Transforming such a probability into a Gaussian is a difficult task, especially with such hard borders. Dequantization has therefore been extended to more sophisticated, learnable distributions beyond uniform in a variational framework. In particular, if we remember the learning objective $\log p(x) = \log \mathbb{E}_u \left[\frac{p(x+u)}{q(u|x)} \right]$, the uniform distribution can be replaced by a learned distribution $q_\theta(u|x)$ with support over $u \in [0, 1]^D$. This approach is called Variational Dequantization and has been proposed by Ho et al. [3]. How can we learn such a distribution? We can use a second normalizing flow that takes x as external input and learns a flexible distribution over u . To ensure a support over $[0, 1]^D$, we can apply a sigmoid activation function as final flow transformation.

Inheriting the original dequantization class, we can implement variational dequantization as follows:

```
[10]: class VariationalDequantization(Dequantization):

    def __init__(self, var_flows, alpha=1e-5):
        """
        Inputs:
            var_flows - A list of flow transformations to use for modeling q(u/x)
            alpha - Small constant, see Dequantization for details
        """
        super().__init__(alpha=alpha)
        self.flows = nn.ModuleList(var_flows)

    def dequant(self, z, ldj):
```

(continues on next page)

(continued from previous page)

```

z = z.to(torch.float32)
img = (z / 255.0) * 2 - 1 # We condition the flows on x, i.e. the original image

# Prior of u is a uniform distribution as before
# As most flow transformations are defined on [-infinity,+infinity], we apply an
# inverse sigmoid first.
deq_noise = torch.rand_like(z).detach()
deq_noise, ldj = self.sigmoid(deq_noise, ldj, reverse=True)
for flow in self.flows:
    deq_noise, ldj = flow(deq_noise, ldj, reverse=False, orig_img=img)
    deq_noise, ldj = self.sigmoid(deq_noise, ldj, reverse=False)

# After the flows, apply u as in standard dequantization
z = (z + deq_noise) / 256.0
ldj -= np.log(256.0) * np.prod(z.shape[1:])
return z, ldj

```

Variational dequantization can be used as a substitute for dequantization. We will compare dequantization and variational dequantization in later experiments.

Coupling layers

Next, we look at possible transformations to apply inside the flow. A recent popular flow layer, which works well in combination with deep neural networks, is the coupling layer introduced by Dinh et al. [1]. The input z is arbitrarily split into two parts, $z_{1:j}$ and $z_{j+1:d}$, of which the first remains unchanged by the flow. Yet, $z_{1:j}$ is used to parameterize the transformation for the second part, $z_{j+1:d}$. Various transformations have been proposed in recent time [3,4], but here we will settle for the simplest and most efficient one: affine coupling. In this coupling layer, we apply an affine transformation by shifting the input by a bias μ and scale it by σ . In other words, our transformation looks as follows:

$$z'_{j+1:d} = \mu_\theta(z_{1:j}) + \sigma_\theta(z_{1:j}) \odot z_{j+1:d}$$

The functions μ and σ are implemented as a shared neural network, and the sum and multiplication are performed element-wise. The LDJ is thereby the sum of the logs of the scaling factors: $\sum_i [\log \sigma_\theta(z_{1:j})]_i$. Inverting the layer can as simply be done as subtracting the bias and dividing by the scale:

$$z_{j+1:d} = (z'_{j+1:d} - \mu_\theta(z_{1:j})) / \sigma_\theta(z_{1:j})$$

We can also visualize the coupling layer in form of a computation graph, where z_1 represents $z_{1:j}$, and z_2 represents $z_{j+1:d}$:

In our implementation, we will realize the splitting of variables as masking. The variables to be transformed, $z_{j+1:d}$, are masked when passing z to the shared network to predict the transformation parameters. When applying the transformation, we mask the parameters for $z_{1:j}$ so that we have an identity operation for those variables:

```

[11]: class CouplingLayer(nn.Module):

    def __init__(self, network, mask, c_in):
        """
        Coupling layer inside a normalizing flow.
        Inputs:
            network - A PyTorch nn.Module constituting the deep neural network for mu_
        and sigma.

```

(continues on next page)

(continued from previous page)

```

    Output shape should be twice the channel size as the input.
mask - Binary mask (0 or 1) where 0 denotes that the element should be
→transformed,
    while 1 means the latent will be used as input to the NN.
c_in - Number of input channels
"""

super().__init__()
self.network = network
self.scaling_factor = nn.Parameter(torch.zeros(c_in))
# Register mask as buffer as it is a tensor which is not a parameter,
# but should be part of the modules state.
self.register_buffer('mask', mask)

def forward(self, z, ldj, reverse=False, orig_img=None):
"""
Inputs:
    z - Latent input to the flow
    ldj - The current ldj of the previous flows.
        The ldj of this layer will be added to this tensor.
    reverse - If True, we apply the inverse of the layer.
    orig_img (optional) - Only needed in VarDeq. Allows external
        input to condition the flow on (e.g. original image)
"""
# Apply network to masked input
z_in = z * self.mask
if orig_img is None:
    nn_out = self.network(z_in)
else:
    nn_out = self.network(torch.cat([z_in, orig_img], dim=1))
s, t = nn_out.chunk(2, dim=1)

# Stabilize scaling output
s_fac = self.scaling_factor.exp().view(1, -1, 1, 1)
s = torch.tanh(s / s_fac) * s_fac

# Mask outputs (only transform the second part)
s = s * (1 - self.mask)
t = t * (1 - self.mask)

# Affine transformation
if not reverse:
    # Whether we first shift and then scale, or the other way round,
    # is a design choice, and usually does not have a big impact
    z = (z + t) * torch.exp(s)
    ldj += s.sum(dim=[1, 2, 3])
else:
    z = (z * torch.exp(-s)) - t
    ldj -= s.sum(dim=[1, 2, 3])

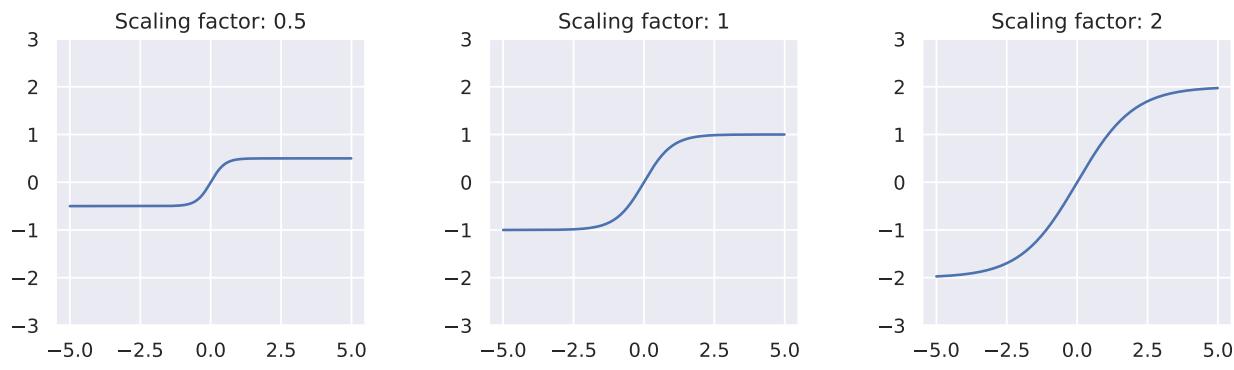
return z, ldj

```

For stabilization purposes, we apply a tanh activation function on the scaling output. This prevents sudden large output values for the scaling that can destabilize training. To still allow scaling factors smaller or larger than -1 and 1

respectively, we have a learnable parameter per dimension, called `scaling_factor`. This scales the tanh to different limits. Below, we visualize the effect of the scaling factor on the output activation of the scaling terms:

```
[12]: with torch.no_grad():
    x = torch.arange(-5, 5, 0.01)
    scaling_factors = [0.5, 1, 2]
    sns.set()
    fig, ax = plt.subplots(1, 3, figsize=(12, 3))
    for i, scale in enumerate(scaling_factors):
        y = torch.tanh(x / scale) * scale
        ax[i].plot(x.numpy(), y.numpy())
        ax[i].set_title("Scaling factor: " + str(scale))
        ax[i].set_ylim(-3, 3)
    plt.subplots_adjust(wspace=0.4)
    sns.reset_orig()
    plt.show()
```



Coupling layers generalize to any masking technique we could think of. However, the most common approach for images is to split the input z in half, using a checkerboard mask or channel mask. A checkerboard mask splits the variables across the height and width dimensions and assigns each other pixel to $z_{j+1:d}$. Thereby, the mask is shared across channels. In contrast, the channel mask assigns half of the channels to $z_{j+1:d}$, and the other half to $z_{1:j+1}$. Note that when we apply multiple coupling layers, we invert the masking for each other layer so that each variable is transformed a similar amount of times.

Let's implement a function that creates a checkerboard mask and a channel mask for us:

```
[13]: def create_checkerboard_mask(h, w, invert=False):
    x, y = torch.arange(h, dtype=torch.int32), torch.arange(w, dtype=torch.int32)
    xx, yy = torch.meshgrid(x, y, indexing='ij')
    mask = torch.fmod(xx + yy, 2)
    mask = mask.to(torch.float32).view(1, 1, h, w)
    if invert:
        mask = 1 - mask
    return mask

def create_channel_mask(c_in, invert=False):
    mask = torch.cat([torch.ones(c_in//2, dtype=torch.float32),
                     torch.zeros(c_in-c_in//2, dtype=torch.float32)])
    mask = mask.view(1, c_in, 1, 1)
    if invert:
        mask = 1 - mask
    return mask
```

We can also visualize the corresponding masks for an image of size $8 \times 8 \times 2$ (2 channels):

```
[14]: checkerboard_mask = create_checkerboard_mask(h=8, w=8).expand(-1,2,-1,-1)
channel_mask = create_channel_mask(c_in=2).expand(-1,-1,8,8)

show_imgs(checkerboard_mask.transpose(0,1), "Checkerboard mask")
show_imgs(channel_mask.transpose(0,1), "Channel mask")

/home/phillip/anaconda3/envs/dl2020/lib/python3.7/site-packages/torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming release, it will be required to pass the indexing argument. (Triggered internally at /opt/conda/conda-bld/pytorch-1646755953518/work/aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs) # type: ignore[attr-defined]
```

Checkerboard mask



Channel mask



As a last aspect of coupling layers, we need to decide for the deep neural network we want to apply in the coupling layers. The input to the layers is an image, and hence we stick with a CNN. Because the input to a transformation depends on all transformations before, it is crucial to ensure a good gradient flow through the CNN back to the input, which can be optimally achieved by a ResNet-like architecture. Specifically, we use a Gated ResNet that adds a σ -gate to the skip connection, similarly to the input gate in LSTMs. The details are not necessarily important here, and the network is strongly inspired from Flow++ [3] in case you are interested in building even stronger models.

```
[15]: class ConcatELU(nn.Module):
    """
    Activation function that applies ELU in both direction (inverted and plain).
    Allows non-linearity while providing strong gradients for any input (important for
    final convolution)
    """

    def forward(self, x):
        return torch.cat([F.elu(x), F.elu(-x)], dim=1)

class LayerNormChannels(nn.Module):
```

(continues on next page)

(continued from previous page)

```

def __init__(self, c_in, eps=1e-5):
    """
    This module applies layer norm across channels in an image.
    Inputs:
        c_in - Number of channels of the input
        eps - Small constant to stabilize std
    """
    super().__init__()
    self.gamma = nn.Parameter(torch.ones(1, c_in, 1, 1))
    self.beta = nn.Parameter(torch.zeros(1, c_in, 1, 1))
    self.eps = eps

def forward(self, x):
    mean = x.mean(dim=1, keepdim=True)
    var = x.var(dim=1, unbiased=False, keepdim=True)
    y = (x - mean) / torch.sqrt(var + self.eps)
    y = y * self.gamma + self.beta
    return y

class GatedConv(nn.Module):
    def __init__(self, c_in, c_hidden):
        """
        This module applies a two-layer convolutional ResNet block with input gate
        Inputs:
            c_in - Number of channels of the input
            c_hidden - Number of hidden dimensions we want to model (usually similar to
        ↪c_in)
        """
        super().__init__()
        self.net = nn.Sequential(
            ConcatELU(),
            nn.Conv2d(2*c_in, c_hidden, kernel_size=3, padding=1),
            ConcatELU(),
            nn.Conv2d(2*c_hidden, 2*c_in, kernel_size=1)
        )

    def forward(self, x):
        out = self.net(x)
        val, gate = out.chunk(2, dim=1)
        return x + val * torch.sigmoid(gate)

class GatedConvNet(nn.Module):
    def __init__(self, c_in, c_hidden=32, c_out=-1, num_layers=3):
        """
        Module that summarizes the previous blocks to a full convolutional neural
        ↪network.
        Inputs:
            c_in - Number of input channels
        """

```

(continues on next page)

(continued from previous page)

```

c_hidden - Number of hidden dimensions to use within the network
c_out - Number of output channels. If -1, 2 times the input channels are used (affine coupling)
num_layers - Number of gated ResNet blocks to apply
"""

super().__init__()
c_out = c_out if c_out > 0 else 2 * c_in
layers = []
layers += [nn.Conv2d(c_in, c_hidden, kernel_size=3, padding=1)]
for layer_index in range(num_layers):
    layers += [GatedConv(c_hidden, c_hidden),
               LayerNormChannels(c_hidden)]
layers += [ConcatELU(),
           nn.Conv2d(2*c_hidden, c_out, kernel_size=3, padding=1)]
self.nn = nn.Sequential(*layers)

self.nn[-1].weight.data.zero_()
self.nn[-1].bias.data.zero_()

def forward(self, x):
    return self.nn(x)

```

Training loop

Finally, we can add Dequantization, Variational Dequantization and Coupling Layers together to build our full normalizing flow on MNIST images. We apply 8 coupling layers in the main flow, and 4 for variational dequantization if applied. We apply a checkerboard mask throughout the network as with a single channel (black-white images), we cannot apply channel mask. The overall architecture is visualized below.

```
[16]: def create_simple_flow(use_vardeq=True):
    flow_layers = []
    if use_vardeq:
        vardeq_layers = [CouplingLayer(network=GatedConvNet(c_in=2, c_out=2, c_hidden=16),
                                     mask=create_checkerboard_mask(h=28, w=28, invert=(i%2==1)),
                         c_in=1) for i in range(4)]
        flow_layers += [VariationalDequantization(var_flows=vardeq_layers)]
    else:
        flow_layers += [Dequantization()]

    for i in range(8):
        flow_layers += [CouplingLayer(network=GatedConvNet(c_in=1, c_hidden=32),
                                     mask=create_checkerboard_mask(h=28, w=28, invert=(i%2==1)),
                         c_in=1)]

    flow_model = ImageFlow(flow_layers).to(device)
    return flow_model
```

For implementing the training loop, we use the framework of PyTorch Lightning and reduce the code overhead. If interested, you can take a look at the generated tensorboard file, in particularly the graph to see an overview of flow transformations that are applied. Note that we again provide pre-trained models (see later on in the notebook) as normalizing flows are particularly expensive to train. We have also run validation and testing as this can take some time as well with the added importance sampling.

```
[17]: def train_flow(flow, model_name="MNISTFlow"):
    # Create a PyTorch Lightning trainer
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, model_name),
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=200,
                          gradient_clip_val=1.0,
                          callbacks=[ModelCheckpoint(save_weights_only=True, mode="min",
                                                      monitor="val_bpd"),
                                     LearningRateMonitor("epoch")],
                          check_val_every_n_epoch=5)
    trainer.logger._log_graph = True
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need

    train_data_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_
                                         last=True, pin_memory=True, num_workers=8)
    result = None

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, model_name + ".ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        ckpt = torch.load(pretrained_filename, map_location=device)
        flow.load_state_dict(ckpt['state_dict'])
        result = ckpt.get("result", None)
    else:
        print("Start training", model_name)
        trainer.fit(flow, train_data_loader, val_loader)

    # Test best model on validation and test set if no result has been found
    # Testing can be expensive due to the importance sampling.
    if result is None:
        val_result = trainer.test(flow, val_loader, verbose=False)
        start_time = time.time()
        test_result = trainer.test(flow, test_loader, verbose=False)
        duration = time.time() - start_time
        result = {"test": test_result, "val": val_result, "time": duration / len(test_
                                         loader) / flow.import_samples}

    return flow, result
```

4.14.3 Multi-scale architecture

One disadvantage of normalizing flows is that they operate on the exact same dimensions as the input. If the input is high-dimensional, so is the latent space, which requires larger computational cost to learn suitable transformations. However, particularly in the image domain, many pixels contain less information in the sense that we could remove them without loosing the semantical information of the image.

Based on this intuition, deep normalizing flows on images commonly apply a multi-scale architecture [1]. After the first N flow transformations, we split off half of the latent dimensions and directly evaluate them on the prior. The other half is run through N more flow transformations, and depending on the size of the input, we split it again in half or stop overall at this position. The two operations involved in this setup is `Squeeze` and `Split` which we will review more closely and implement below.

Squeeze and Split

When we want to remove half of the pixels in an image, we have the problem of deciding which variables to cut, and how to rearrange the image. Thus, the squeezing operation is commonly used before split, which divides the image into subsquares of shape $2 \times 2 \times C$, and reshapes them into $1 \times 1 \times 4C$ blocks. Effectively, we reduce the height and width of the image by a factor of 2 while scaling the number of channels by 4. Afterwards, we can perform the split operation over channels without the need of rearranging the pixels. The smaller scale also makes the overall architecture more efficient. Visually, the squeeze operation should transform the input as follows:

The input of $4 \times 4 \times 1$ is scaled to $2 \times 2 \times 4$ following the idea of grouping the pixels in $2 \times 2 \times 1$ subsquares. Next, let's try to implement this layer:

```
[18]: class SqueezeFlow(nn.Module):

    def forward(self, z, ldj, reverse=False):
        B, C, H, W = z.shape
        if not reverse:
            # Forward direction: H x W x C => H/2 x W/2 x 4C
            z = z.reshape(B, C, H//2, 2, W//2, 2)
            z = z.permute(0, 1, 3, 5, 2, 4)
            z = z.reshape(B, 4*C, H//2, W//2)
        else:
            # Reverse direction: H/2 x W/2 x 4C => H x W x C
            z = z.reshape(B, C//4, 2, 2, H, W)
            z = z.permute(0, 1, 4, 2, 5, 3)
            z = z.reshape(B, C//4, H*2, W*2)
        return z, ldj
```

Before moving on, we can verify our implementation by comparing our output with the example figure above:

```
[19]: sq_flow = SqueezeFlow()
rand_img = torch.arange(1,17).view(1, 1, 4, 4)
print("Image (before)\n", rand_img)
forward_img, _ = sq_flow(rand_img, ldj=None, reverse=False)
print("\nImage (forward)\n", forward_img.permute(0,2,3,1)) # Permute for readability
reconst_img, _ = sq_flow(forward_img, ldj=None, reverse=True)
print("\nImage (reverse)\n", reconst_img)

Image (before)
tensor([[[[ 1,  2,  3,  4],
```

(continues on next page)

(continued from previous page)

```
[ 5,  6,  7,  8],
[ 9, 10, 11, 12],
[13, 14, 15, 16]]]))
```

```
Image (forward)
tensor([[[[ 1,  2,  5,  6],
          [ 3,  4,  7,  8]],

         [[ 9, 10, 13, 14],
          [11, 12, 15, 16]]]])
```

```
Image (reverse)
tensor([[[[ 1,  2,  3,  4],
          [ 5,  6,  7,  8],
          [ 9, 10, 11, 12],
          [13, 14, 15, 16]]]])
```

The split operation divides the input into two parts, and evaluates one part directly on the prior. So that our flow operation fits to the implementation of the previous layers, we will return the prior probability of the first part as the log determinant jacobian of the layer. It has the same effect as if we would combine all variable splits at the end of the flow, and evaluate them together on the prior.

```
[20]: class SplitFlow(nn.Module):

    def __init__(self):
        super().__init__()
        self.prior = torch.distributions.normal.Normal(loc=0.0, scale=1.0)

    def forward(self, z, ldj, reverse=False):
        if not reverse:
            z, z_split = z.chunk(2, dim=1)
            ldj += self.prior.log_prob(z_split).sum(dim=[1,2,3])
        else:
            z_split = self.prior.sample(sample_shape=z.shape).to(device)
            z = torch.cat([z, z_split], dim=1)
            ldj -= self.prior.log_prob(z_split).sum(dim=[1,2,3])
        return z, ldj
```

Building a multi-scale flow

After defining the squeeze and split operation, we are finally able to build our own multi-scale flow. Deep normalizing flows such as Glow and Flow++ [2,3] often apply a split operation directly after squeezing. However, with shallow flows, we need to be more thoughtful about where to place the split operation as we need at least a minimum amount of transformations on each variable. Our setup is inspired by the original RealNVP architecture [1] which is shallower than other, more recent state-of-the-art architectures.

Hence, for the MNIST dataset, we will apply the first squeeze operation after two coupling layers, but don't apply a split operation yet. Because we have only used two coupling layers and each the variable has been only transformed once, a split operation would be too early. We apply two more coupling layers before finally applying a split flow and squeeze again. The last four coupling layers operate on a scale of $7 \times 7 \times 8$. The full flow architecture is shown below.

Note that while the feature maps inside the coupling layers reduce with the height and width of the input, the increased number of channels is not directly considered. To counteract this, we increase the hidden dimensions for the coupling layers on the squeezed input. The dimensions are often scaled by 2 as this approximately increases the computation cost by 4 canceling with the squeezing operation. However, we will choose the hidden dimensionalities 32, 48, 64 for the three scales respectively to keep the number of parameters reasonable and show the efficiency of multi-scale architectures.

```
[21]: def create_multiscale_flow():
    flow_layers = []

    vardeq_layers = [CouplingLayer(network=GatedConvNet(c_in=2, c_out=2, c_hidden=16),
                                    mask=create_checkerboard_mask(h=28, w=28, invert=(i
                                    ↵%2==1)),
                      c_in=1) for i in range(4)]
    flow_layers += [VariationalDequantization(vardeq_layers)]

    flow_layers += [CouplingLayer(network=GatedConvNet(c_in=1, c_hidden=32),
                                    mask=create_checkerboard_mask(h=28, w=28, invert=(i
                                    ↵%2==1)),
                      c_in=1) for i in range(2)]
    flow_layers += [SqueezeFlow()]
    for i in range(2):
        flow_layers += [CouplingLayer(network=GatedConvNet(c_in=4, c_hidden=48),
                                    mask=create_channel_mask(c_in=4, invert=(i%2==1)),
                                    c_in=4)]
    flow_layers += [SplitFlow(),
                  SqueezeFlow()]
    for i in range(4):
        flow_layers += [CouplingLayer(network=GatedConvNet(c_in=8, c_hidden=64),
                                    mask=create_channel_mask(c_in=8, invert=(i%2==1)),
                                    c_in=8)]

    flow_model = ImageFlow(flow_layers).to(device)
    return flow_model
```

We can show the difference in number of parameters below:

```
[22]: def print_num_params(model):
    num_params = sum([np.prod(p.shape) for p in model.parameters()])
    print("Number of parameters: {:.3f}".format(num_params))

print_num_params(create_simple_flow(use_vardeq=False))
print_num_params(create_simple_flow(use_vardeq=True))
print_num_params(create_multiscale_flow())

Number of parameters: 556,312
Number of parameters: 628,388
Number of parameters: 1,711,818
```

Although the multi-scale flow has almost 3 times the parameters of the single scale flow, it is not necessarily more computationally expensive than its counterpart. We will compare the runtime in the following experiments as well.

4.14.4 Analysing the flows

In the last part of the notebook, we will train all the models we have implemented above, and try to analyze the effect of the multi-scale architecture and variational dequantization.

Training flow variants

Before we can analyse the flow models, we need to train them first. We provide pre-trained models that contain the validation and test performance, and run-time information. As flow models are computationally expensive, we advice you to rely on those pretrained models for a first run through the notebook.

```
[23]: flow_dict = {"simple": {}, "vardeq": {}, "multiscale": {}}
flow_dict["simple"]["model"], flow_dict["simple"]["result"] = train_flow(create_simple_
    ↴flow(use_vardeq=False), model_name="MNISTFlow_simple")
flow_dict["vardeq"]["model"], flow_dict["vardeq"]["result"] = train_flow(create_simple_
    ↴flow(use_vardeq=True), model_name="MNISTFlow_vardeq")
flow_dict["multiscale"]["model"], flow_dict["multiscale"]["result"] = train_flow(create_
    ↴multiscale_flow(), model_name="MNISTFlow_multiscale")

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

Found pretrained model, loading...
Found pretrained model, loading...

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

Found pretrained model, loading...
```

```
[24]: flow_dict["simple"]["result"]

[24]: {'test': [{'test_bpd': 1.07839834690094}],
       'val': [{"test_bpd": 1.0798484086990356}],
       'time': 0.019570968523147,
       'samp_time': 0.017716965675354003}
```

Density modeling and sampling

Firstly, we can compare the models on their quantitative results. The following table shows all important statistics. The inference time specifies the time needed to determine the probability for a batch of 64 images for each model, and the sampling time the duration it took to sample a batch of 64 images.

```
[25]: %%html
<!-- Some HTML code to increase font size in the following table -->
<style>
```

(continues on next page)

(continued from previous page)

```
th {font-size: 120%;}
td {font-size: 120%;}
</style>
```

```
<IPython.core.display.HTML object>
```

```
[26]: import tabulate
from IPython.display import display, HTML

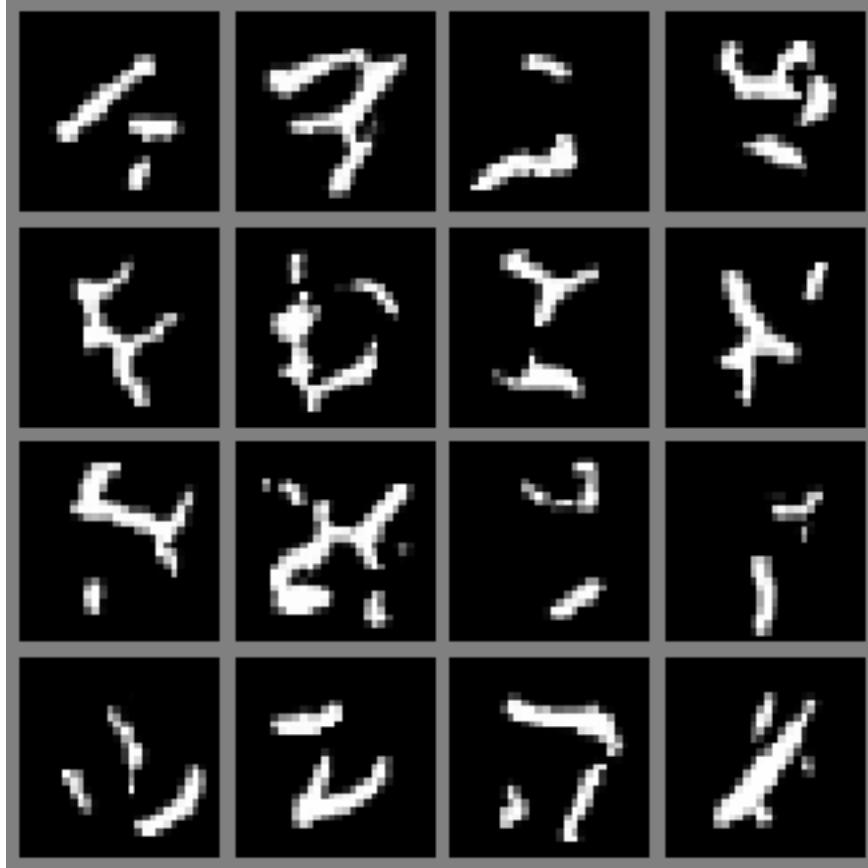
table = [[key,
          "%4.3f bpd" % flow_dict[key]["result"]["val"][0]["test_bpd"],
          "%4.3f bpd" % flow_dict[key]["result"]["test"][0]["test_bpd"],
          "%2.0f ms" % (1000 * flow_dict[key]["result"]["time"]),
          "%2.0f ms" % (1000 * flow_dict[key]["result"].get("samp_time", 0)),
          "{}".format(sum([np.prod(p.shape) for p in flow_dict[key]["model"]].
          ↪parameters()])],
          for key in flow_dict]
display(HTML(tabulate.tabulate(table, tablefmt='html', headers=["Model", "Validation Bpd
          ↪", "Test Bpd", "Inference time", "Sampling time", "Num Parameters"])))
<IPython.core.display.HTML object>
```

As we have initially expected, using variational dequantization improves upon standard dequantization in terms of bits per dimension. Although the difference with 0.04bpd doesn't seem impressive first, it is a considerable step for generative models (most state-of-the-art models improve upon previous models in a range of 0.02-0.1bpd on CIFAR with three times as high bpd). While it takes longer to evaluate the probability of an image due to the variational dequantization, which also leads to a longer training time, it does not have an effect on the sampling time. This is because inverting variational dequantization is the same as dequantization: finding the next lower integer.

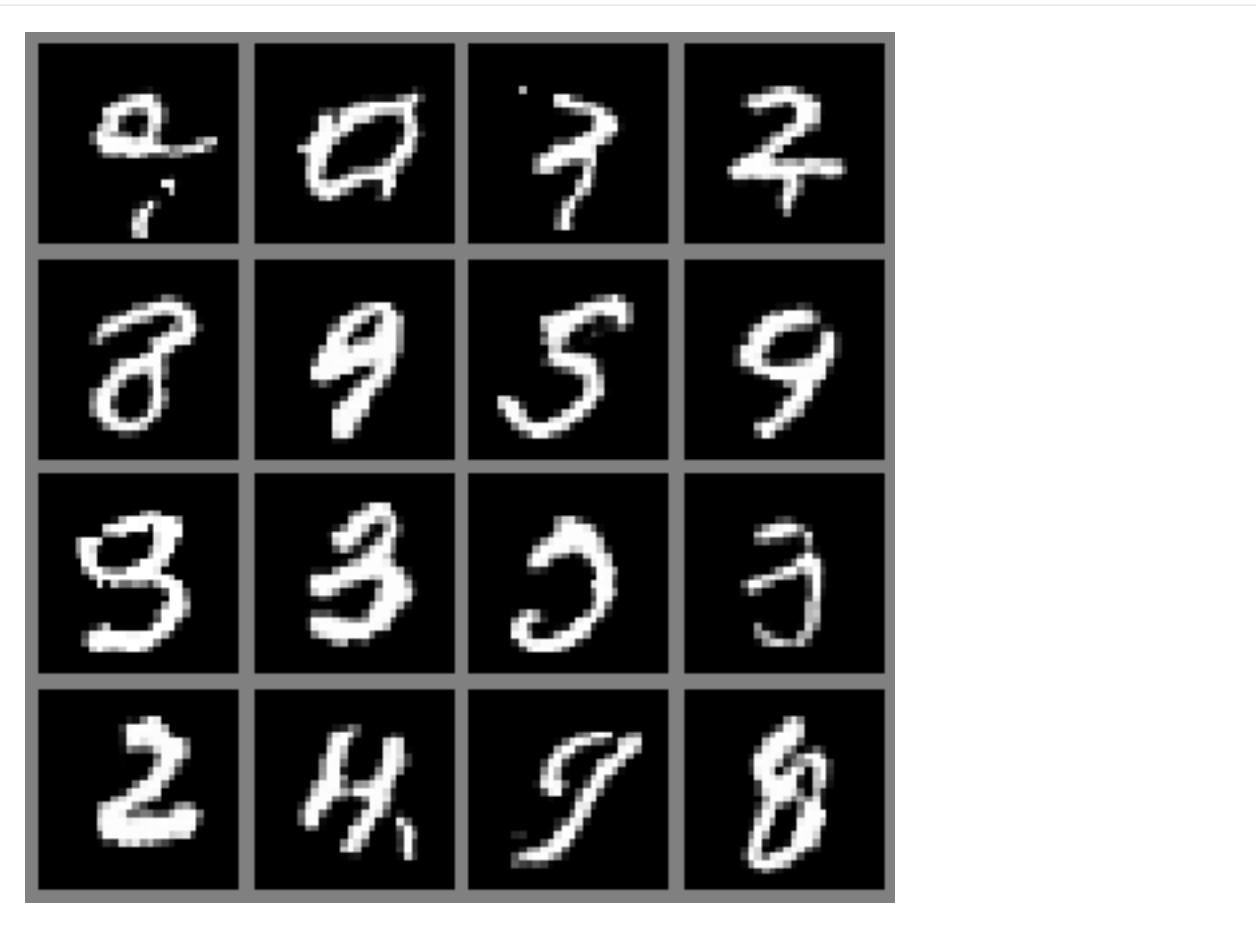
When we compare the two models to multi-scale architecture, we can see that the bits per dimension score again dropped by about 0.02bpd. Additionally, the sampling time improved notably despite having more parameters. Thus, we see that the multi-scale flow is not only stronger for density modeling, but also more efficient.

Next, we can test the sampling quality of the models. We should note that the samples for variational dequantization and standard dequantization are very similar, and hence we visualize here only the ones for variational dequantization and the multi-scale model. However, feel free to also test out the "simple" model. The seeds are set to obtain reproducible generations and are not cherry picked.

```
[27]: pl.seed_everything(44)
samples = flow_dict["vardeq"]["model"].sample(img_shape=[16, 1, 28, 28])
show_imgs(samples.cpu())
Global seed set to 44
```



```
[28]: pl.seed_everything(42)
samples = flow_dict["multiscale"]["model"].sample(img_shape=[16,8,7,7])
show_imgs(samples.cpu())
Global seed set to 42
```



From the few samples, we can see a clear difference between the simple and the multi-scale model. The single-scale model has only learned local, small correlations while the multi-scale model was able to learn full, global relations that form digits. This shows another benefit of the multi-scale model. In contrast to VAEs, the outputs are sharp as normalizing flows can naturally model complex, multi-modal distributions while VAEs have the independent decoder output noise. Nevertheless, the samples from this flow are far from perfect as not all samples show true digits.

Interpolation in latent space

Another popular test for the smoothness of the latent space of generative models is to interpolate between two training examples. As normalizing flows are strictly invertible, we can guarantee that any image is represented in the latent space. We again compare the variational dequantization model with the multi-scale model below.

```
[29]: @torch.no_grad()
def interpolate(model, img1, img2, num_steps=8):
    """
    Inputs:
        model - object of ImageFlow class that represents the (trained) flow model
        img1, img2 - Image tensors of shape [1, 28, 28]. Images between which should be
        ↪ interpolated.
        num_steps - Number of interpolation steps. 8 interpolation steps mean 6
        ↪ intermediate pictures besides img1 and img2
    """
    imgs = torch.stack([img1, img2], dim=0).to(model.device)
```

(continues on next page)

(continued from previous page)

```

z, _ = model.encode(imgs)
alpha = torch.linspace(0, 1, steps=num_steps, device=z.device).view(-1, 1, 1, 1)
interpolations = z[0:1] * alpha + z[1:2] * (1 - alpha)
interp_imgs = model.sample(interpolations.shape[:1] + imgs.shape[1:], z_
→init=interpolations)
show_imgs(interp_imgs, row_size=8)

exmp_imgs, _ = next(iter(train_loader))

```

[30]:

```

pl.seed_everything(42)
for i in range(2):
    interpolate(flow_dict["vardeq"]["model"], exmp_imgs[2*i], exmp_imgs[2*i+1])

```

Global seed set to 42



[31]:

```

pl.seed_everything(42)
for i in range(2):
    interpolate(flow_dict["multiscale"]["model"], exmp_imgs[2*i], exmp_imgs[2*i+1])

```

Global seed set to 42



The interpolations of the multi-scale model result in more realistic digits (first row 7 \leftrightarrow 8 \leftrightarrow 6, second row 9 \leftrightarrow 6), while the variational dequantization model focuses on local patterns that globally do not form a digit. For the multi-scale model, we actually did not do the “true” interpolation between the two images as we did not consider the variables that were split along the flow (they have been sampled randomly for all samples). However, as we will see in the next experiment, the early variables do not effect the overall image much.

Visualization of latents in different levels of multi-scale

In the following we will focus more on the multi-scale flow. We want to analyse what information is being stored in the variables split at early layers, and what information for the final variables. For this, we sample 8 images where each of them share the same final latent variables, but differ in the other part of the latent variables. Below we visualize three examples of this:

```
[32]: pl.seed_everything(44)
for _ in range(3):
    z_init = flow_dict["multiscale"]["model"].prior.sample(sample_shape=[1, 8, 7, 7])
    z_init = z_init.expand(8, -1, -1, -1)
    samples = flow_dict["multiscale"]["model"].sample(img_shape=z_init.shape, z_init=z_
    ~init)
    show_imgs(samples.cpu())
```

Global seed set to 44





We see that the early split variables indeed have a smaller effect on the image. Still, small differences can be spot when we look carefully at the borders of the digits. For instance, in the middle, the top part of the 3 has different thicknesses for different samples although all of them represent the same coarse structure. This shows that the flow indeed learns to separate the higher-level information in the final variables, while the early split ones contain local noise patterns.

Visualizing Dequantization

As a final part of this notebook, we will look at the effect of variational dequantization. We have motivated variational dequantization by the issue of sharp edges/boarders being difficult to model, and a flow would rather prefer smooth, prior-like distributions. To check how what noise distribution $q(u|x)$ the flows in the variational dequantization module have learned, we can plot a histogram of output values from the dequantization and variational dequantization module.

```
[33]: def visualize_dequant_distribution(model : ImageFlow, imgs : torch.Tensor, title:
    ↪str=None):
    """
    Inputs:
        model - The flow of which we want to visualize the dequantization distribution
        imgs - Example training images of which we want to visualize the dequantization
    ↪distribution
    """
    imgs = imgs.to(device)
    ldj = torch.zeros(imgs.shape[0], dtype=torch.float32).to(device)
    with torch.no_grad():
        dequant_vals = []
        for _ in tqdm(range(8), leave=False):
            d, _ = model.flows[0](imgs, ldj, reverse=False)
            dequant_vals.append(d)
        dequant_vals = torch.cat(dequant_vals, dim=0)
    dequant_vals = dequant_vals.view(-1).cpu().numpy()
    sns.set()
    plt.figure(figsize=(10,3))
    plt.hist(dequant_vals, bins=256, color=to_rgb("C0")+(0.5,), edgecolor="C0",
    ↪density=True)
    if title is not None:
        plt.title(title)
    plt.show()
```

(continues on next page)

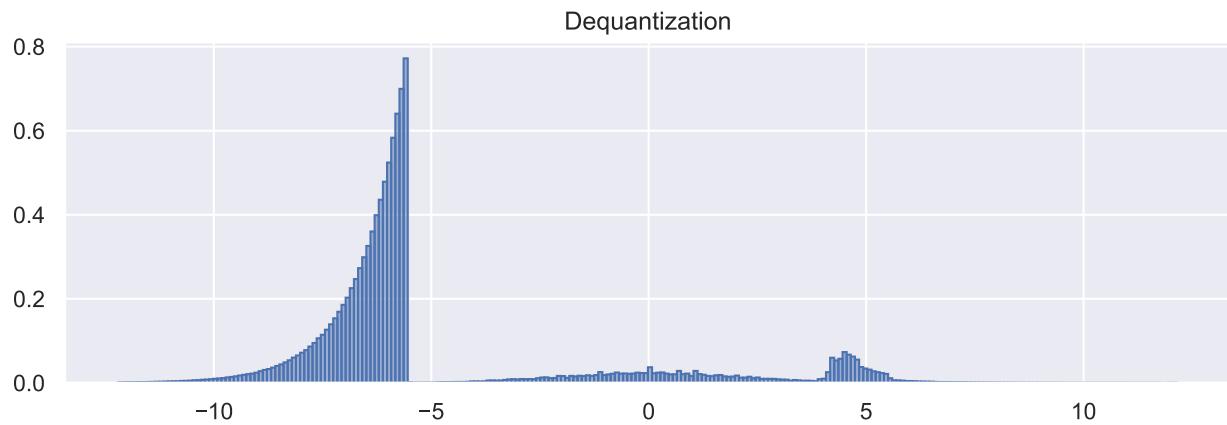
(continued from previous page)

```
plt.close()

sample_imgs, _ = next(iter(train_loader))
```

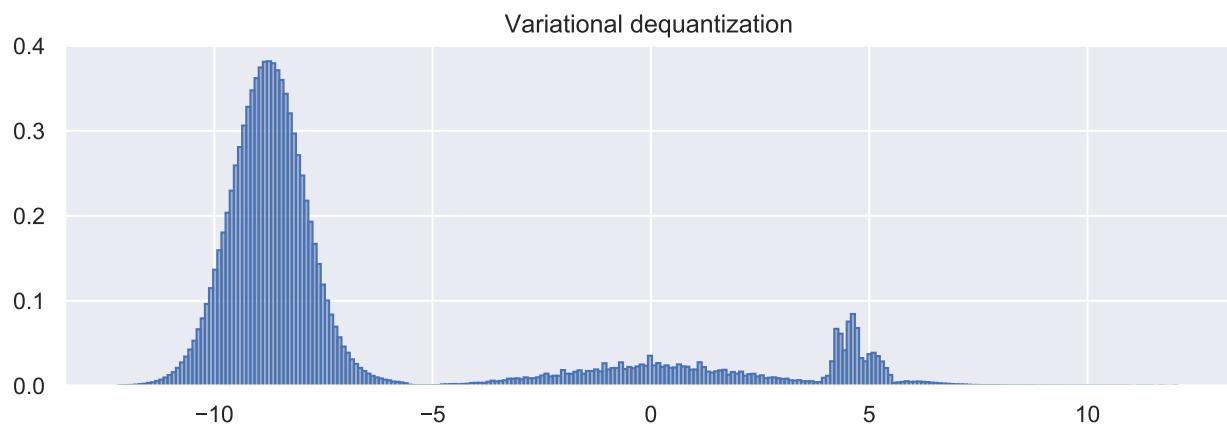
[34]: `visualize_dequant_distribution(flow_dict["simple"]["model"], sample_imgs, title=`
`↪ "Dequantization")`

0% | 0/8 [00:00<?, ?it/s]



[35]: `visualize_dequant_distribution(flow_dict["vardeq"]["model"], sample_imgs, title=`
`↪ "Variational dequantization")`

0% | 0/8 [00:00<?, ?it/s]



The dequantization distribution in the first plot shows that the MNIST images have a strong bias towards 0 (black), and the distribution of them have a sharp border as mentioned before. The variational dequantization module has indeed learned a much smoother distribution with a Gaussian-like curve which can be modeled much better. For the other values, we would need to visualize the distribution $q(u|x)$ on a deeper level, depending on x . However, as all u 's interact and depend on each other, we would need to visualize a distribution in 784 dimensions, which is not that intuitive anymore.

4.14.5 Conclusion

In conclusion, we have seen how to implement our own normalizing flow, and what difficulties arise if we want to apply them on images. Dequantization is a crucial step in mapping the discrete images into continuous space to prevent undesirable delta-peak solutions. While dequantization creates hypercubes with hard border, variational dequantization allows us to fit a flow much better on the data. This allows us to obtain a lower bits per dimension score, while not affecting the sampling speed. The most common flow element, the coupling layer, is simple to implement, and yet effective. Furthermore, multi-scale architectures help to capture the global image context while allowing us to efficiently scale up the flow. Normalizing flows are an interesting alternative to VAEs as they allow an exact likelihood estimate in continuous space, and we have the guarantee that every possible input x has a corresponding latent vector z . However, even beyond continuous inputs and images, flows can be applied and allow us to exploit the data structure in latent space, as e.g. on graphs for the task of molecule generation [6]. Recent advances in [Neural ODEs](#) allow a flow with infinite number of layers, called Continuous Normalizing Flows, whose potential is yet to fully explore. Overall, normalizing flows are an exciting research area which will continue over the next couple of years.

4.14.6 References

- [1] Dinh, L., Sohl-Dickstein, J., and Bengio, S. (2017). “Density estimation using Real NVP,” In: 5th International Conference on Learning Representations, ICLR 2017. [Link](#)
 - [2] Kingma, D. P., and Dhariwal, P. (2018). “Glow: Generative Flow with Invertible 1x1 Convolutions,” In: Advances in Neural Information Processing Systems, vol. 31, pp. 10215–10224. [Link](#)
 - [3] Ho, J., Chen, X., Srinivas, A., Duan, Y., and Abbeel, P. (2019). “Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design,” in Proceedings of the 36th International Conference on Machine Learning, vol. 97, pp. 2722–2730. [Link](#)
 - [4] Durkan, C., Bekasov, A., Murray, I., and Papamakarios, G. (2019). “Neural Spline Flows,” In: Advances in Neural Information Processing Systems, pp. 7509–7520. [Link](#)
 - [5] Hoogeboom, E., Cohen, T. S., and Tomczak, J. M. (2020). “Learning Discrete Distributions by Dequantization,” arXiv preprint arXiv2001.11235v1. [Link](#)
 - [6] Lippe, P., and Gavves, E. (2021). “Categorical Normalizing Flows via Continuous Transformations,” In: International Conference on Learning Representations, ICLR 2021. [Link](#)
-

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on [GitHub](#).

4.15 Tutorial 12: Autoregressive Image Modeling

Filled notebook:

Pre-trained models:

Recordings:

JAX+Flax version:

Author: Phillip Lippe

Note: Interested in JAX? Check out our [JAX+Flax](#) version of this tutorial!

In this tutorial, we implement an autoregressive likelihood model for the task of image modeling. Autoregressive models are naturally strong generative models that constitute one of the current state-of-the-art architectures on likelihood-based image modeling, and are also the basis for large language generation models such as GPT3. Similar to the language generation you have seen in assignment 2, autoregressive models work on images by modeling the likelihood of a pixel given all previous ones. For instance, in the picture below, we model the pixel x_i as a conditional probability distribution based on all previous (here blue) pixels (figure credit - Aaron van den Oord et al.):

Generally, autoregressive model over high-dimensional data \mathbf{x} factor the joint distribution as the following product of conditionals:

$$p(\mathbf{x}) = p(x_1, \dots, x_n) = \prod_{i=1}^n p(x_i|x_1, \dots, x_{i-1})$$

Learning these conditionals is often much simpler than learning the joint distribution $p(\mathbf{x})$ all together. However, disadvantages of autoregressive models include slow sampling, especially for large images, as we need height-times-width forward passes through the model. In addition, for some applications, we require a latent space as modeled in VAEs and Normalizing Flows. For instance, in autoregressive models, we cannot interpolate between two images because of the lack of a latent representation. We will explore and discuss these benefits and drawbacks alongside with our implementation.

Our implementation will focus on the [PixelCNN](#) [2] model which has been discussed in detail in the lecture. Most current SOTA models use PixelCNN as their fundamental architecture, and various additions have been proposed to improve the performance (e.g. [PixelCNN++](#) and [PixelSNAIL](#)). Hence, implementing PixelCNN is a good starting point for our short tutorial.

First of all, we need to import our standard libraries. Similarly as in the last couple of tutorials, we will use PyTorch Lightning here as well.

```
[1]: ## Standard libraries
import os
import math
import numpy as np

## Imports for plotting
import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import seaborn as sns

## Progress bar
from tqdm.notebook import tqdm

## PyTorch
import torch
```

(continues on next page)

(continued from previous page)

```

import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
# Torchvision
import torchvision
from torchvision.datasets import MNIST
from torchvision import transforms
# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial12"

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Fetching the device that will be used throughout this notebook
device = torch.device("cpu") if not torch.cuda.is_available() else torch.device("cuda:0")
print("Using device", device)
Using device cuda:0

```

We again provide a pretrained model, which is downloaded below:

```

[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial12/"
# Files to download
pretrained_files = ["PixelCNN.ckpt"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:

```

(continues on next page)

(continued from previous page)

```

    urllib.request.urlretrieve(file_url, file_path)
except HTTPError as e:
    print("Something went wrong. Please try to download the file from the GDrive_"
    "folder, or contact the author with the full output including the following error:\n",_
    "e)

```

Similar to the Normalizing Flows in Tutorial 11, we will work on the MNIST dataset and use 8-bits per pixel (values between 0 and 255). The dataset is loaded below:

```
[3]: # Convert images from 0-1 to 0-255 (integers). We use the long datatype as we will use_
# the images as labels as well
def discretize(sample):
    return (sample * 255).to(torch.long)

# Transformations applied on each image => only make them a tensor
transform = transforms.Compose([transforms.ToTensor(),
                                discretize])

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = MNIST(root=DATASET_PATH, train=True, transform=transform, download=True)
pl.seed_everything(42)
train_set, val_set = torch.utils.data.random_split(train_dataset, [50000, 10000])

# Loading the test set
test_set = MNIST(root=DATASET_PATH, train=False, transform=transform, download=True)

# We define a set of data loaders that we can use for various purposes later.
train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True,_
                                pin_memory=True, num_workers=4)
val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False, drop_last=False,_
                                num_workers=4)
test_loader = data.DataLoader(test_set, batch_size=128, shuffle=False, drop_last=False,_
                                num_workers=4)
```

A good practice is to always visualize some data examples to get an intuition of the data:

```
[4]: def show_imgs(imgs):
    num_imgs = imgs.shape[0] if isinstance(imgs, torch.Tensor) else len(imgs)
    nrow = min(num_imgs, 4)
    ncol = int(math.ceil(num_imgs/nrow))
    imgs = torchvision.utils.make_grid(imgs, nrow=nrow, pad_value=128)
    imgs = imgs.clamp(min=0, max=255)
    np_imgs = imgs.cpu().numpy()
    plt.figure(figsize=(1.5*nrow, 1.5*ncol))
    plt.imshow(np.transpose(np_imgs, (1, 2, 0)), interpolation='nearest')
    plt.axis('off')
    plt.show()
    plt.close()

show_imgs([train_set[i][0] for i in range(8)])
```



4.15.1 Masked autoregressive convolutions

The core module of PixelCNN is its masked convolutions. In contrast to language models, we don't apply an LSTM on each pixel one-by-one. This would be inefficient because images are grids instead of sequences. Thus, it is better to rely on convolutions that have shown great success in deep CNN classification models.

Nevertheless, we cannot just apply standard convolutions without any changes. Remember that during training of autoregressive models, we want to use teacher forcing which both helps the model training, and significantly reduces the time needed for training. For image modeling, teacher forcing is implemented by using a training image as input to the model, and we want to obtain as output the prediction for each pixel based on *only* its predecessors. Thus, we need to ensure that the prediction for a specific pixel can only be influenced by its predecessors and not by its own value or any “future” pixels. For this, we apply convolutions with a mask.

Which mask we use depends on the ordering of pixels we decide on, i.e. which is the first pixel we predict, which is the second one, etc. The most commonly used ordering is to denote the upper left pixel as the start pixel, and sort the pixels row by row, as shown in the visualization at the top of the tutorial. Thus, the second pixel is on the right of the first one (first row, second column), and once we reach the end of the row, we start in the second row, first column. If we now want to apply this to our convolutions, we need to ensure that the prediction of pixel 1 is not influenced by its own “true” input, and all pixels on its right and in any lower row. In convolutions, this means that we want to set those entries of the weight matrix to zero that take pixels on the right and below into account. As an example for a 5x5 kernel, see a mask below (figure credit - [Aaron van den Oord](#)):

Before looking into the application of masked convolutions in PixelCNN in detail, let's first implement a module that allows us to apply an arbitrary mask to a convolution:

```
[5]: class MaskedConvolution(nn.Module):
    def __init__(self, c_in, c_out, mask, **kwargs):
        """
        Implements a convolution with mask applied on its weights.
        Inputs:
            c_in - Number of input channels
            c_out - Number of output channels
            mask - Tensor of shape [kernel_size_H, kernel_size_W] with 0s where
                   the convolution should be masked, and 1s otherwise.
            kwargs - Additional arguments for the convolution
        """

```

(continues on next page)

(continued from previous page)

```

"""
super().__init__()
# For simplicity: calculate padding automatically
kernel_size = (mask.shape[0], mask.shape[1])
dilation = 1 if "dilation" not in kwargs else kwargs["dilation"]
padding = tuple([dilation*(kernel_size[i]-1)//2 for i in range(2)])
# Actual convolution
self.conv = nn.Conv2d(c_in, c_out, kernel_size, padding=padding, **kwargs)

# Mask as buffer => it is no parameter but still a tensor of the module
# (must be moved with the devices)
self.register_buffer('mask', mask[None,None])

def forward(self, x):
    self.conv.weight.data *= self.mask # Ensures zero's at masked positions
    return self.conv(x)

```

Vertical and horizontal convolution stacks

To build our own autoregressive image model, we could simply stack a few masked convolutions on top of each other. This was actually the case for the original PixelCNN model, discussed in the paper [Pixel Recurrent Neural Networks](#), but this leads to a considerable issue. When sequentially applying a couple of masked convolutions, the receptive field of a pixel show to have a “blind spot” on the right upper side, as shown in the figure below (figure credit - Aaron van den Oord et al.):

Although a pixel should be able to take into account all other pixels above and left of it, a stack of masked convolutions does not allow us to look to the upper pixels on the right. This is because the features of the pixels above, which we use for convolution, do not contain any information of the pixels on the right of the same row. If they would, we would be “cheating” and actually looking into the future. To overcome this issue, van den Oord et. al [2] proposed to split the convolutions into a vertical and a horizontal stack. The vertical stack looks at all pixels above the current one, while the horizontal takes into account all on the left. While keeping both of them separate, we can actually look at the pixels on the right with the vertical stack without breaking any of our assumptions. The two convolutions are also shown in the figure above.

Let us implement them here as follows:

```
[6]: class VerticalStackConvolution(MaskedConvolution):

    def __init__(self, c_in, c_out, kernel_size=3, mask_center=False, **kwargs):
        # Mask out all pixels below. For efficiency, we could also reduce the kernel
        # size in height, but for simplicity, we stick with masking here.
        mask = torch.ones(kernel_size, kernel_size)
        mask[kernel_size//2+1:,:] = 0

        # For the very first convolution, we will also mask the center row
        if mask_center:
            mask[kernel_size//2,:] = 0

    super().__init__(c_in, c_out, mask, **kwargs)
```

(continues on next page)

(continued from previous page)

```
class HorizontalStackConvolution(MaskedConvolution):

    def __init__(self, c_in, c_out, kernel_size=3, mask_center=False, **kwargs):
        # Mask out all pixels on the left. Note that our kernel has a size of 1
        # in height because we only look at the pixel in the same row.
        mask = torch.ones(1,kernel_size)
        mask[0,kernel_size//2+1:] = 0

        # For the very first convolution, we will also mask the center pixel
        if mask_center:
            mask[0,kernel_size//2] = 0

    super().__init__(c_in, c_out, mask, **kwargs)
```

Note that we have an input argument called `mask_center`. Remember that the input to the model is the actual input image. Hence, the very first convolution we apply cannot use the center pixel as input, but must be masked. All consecutive convolutions, however, should use the center pixel as we otherwise lose the features of the previous layer. Hence, the input argument `mask_center` is True for the very first convolutions, and False for all others.

Visualizing the receptive field

To validate our implementation of masked convolutions, we can visualize the receptive field we obtain with such convolutions. We should see that with increasing number of convolutional layers, the receptive field grows in both vertical and horizontal direction, without the issue of a blind spot. The receptive field can be empirically measured by backpropagating an arbitrary loss for the output features of a specific pixel with respect to the input. We implement this idea below, and visualize the receptive field below.

```
[7]: inp_img = torch.zeros(1, 1, 11, 11)
inp_img.requires_grad_()

def show_center_recep_field(img, out):
    """
    Calculates the gradients of the input with respect to the output center pixel,
    and visualizes the overall receptive field.
    Inputs:
        img - Input image for which we want to calculate the receptive field on.
        out - Output features/loss which is used for backpropagation, and should be
              the output of the network/computation graph.
    """
    # Determine gradients
    loss = out[0,:,:img.shape[2]//2,img.shape[3]//2].sum() # L1 loss for simplicity
    loss.backward(retain_graph=True) # Retain graph as we want to stack multiple layers
    # and show the receptive field of all of them
    img_grads = img.grad.abs()
    img.grad.fill_(0) # Reset grads

    # Plot receptive field
    img = img_grads.squeeze().cpu().numpy()
    fig, ax = plt.subplots(1,2)
    pos = ax[0].imshow(img)
    ax[1].imshow(img>0)
```

(continues on next page)

(continued from previous page)

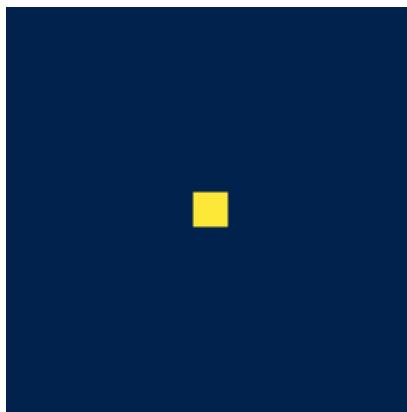
```

# Mark the center pixel in red if it doesn't have any gradients (should be the case for standard autoregressive models)
show_center = (img[img.shape[0]//2,img.shape[1]//2] == 0)
if show_center:
    center_pixel = np.zeros(img.shape + (4,))
    center_pixel[center_pixel.shape[0]//2:center_pixel.shape[1]//2,:] = np.array([1.0, 0.0, 0.0, 1.0])
for i in range(2):
    ax[i].axis('off')
    if show_center:
        ax[i].imshow(center_pixel)
ax[0].set_title("Weighted receptive field")
ax[1].set_title("Binary receptive field")
plt.show()
plt.close()

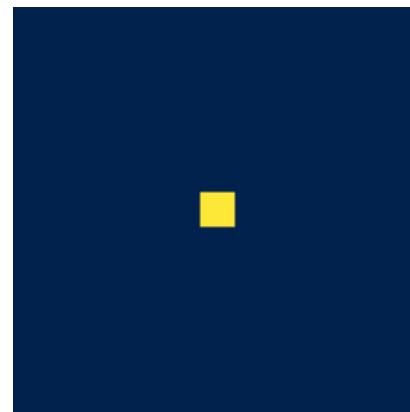
show_center_recep_field(inp_img, inp_img)

```

Weighted receptive field



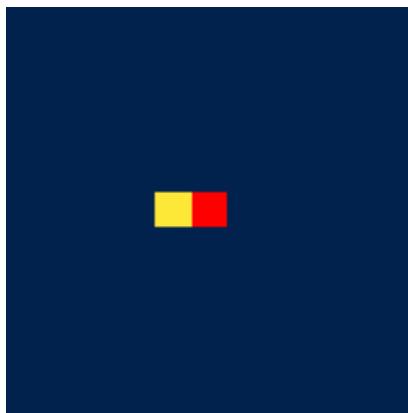
Binary receptive field



Let's first visualize the receptive field of a horizontal convolution without the center pixel. We use a small, arbitrary input image (11×11 pixels), and calculate the loss for the center pixel. For simplicity, we initialize all weights with 1 and the bias with 0, and use a single channel. This is sufficient for our visualization purposes.

```
[8]: horiz_conv = HorizontalStackConvolution(c_in=1, c_out=1, kernel_size=3, mask_center=True)
horiz_conv.conv.weight.data.fill_(1)
horiz_conv.conv.bias.data.fill_(0)
horiz_img = horiz_conv(inp_img)
show_center_recep_field(inp_img, horiz_img)
```

Weighted receptive field



Binary receptive field

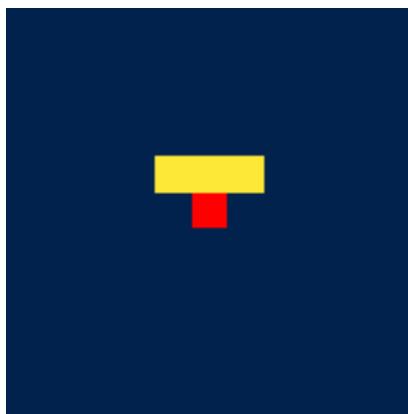


The receptive field is shown in yellow, the center pixel in red, and all other pixels outside of the receptive field are dark blue. As expected, the receptive field of a single horizontal convolution with the center pixel masked and a 3×3 kernel is only the pixel on the left. If we use a larger kernel size, more pixels would be taken into account on the left.

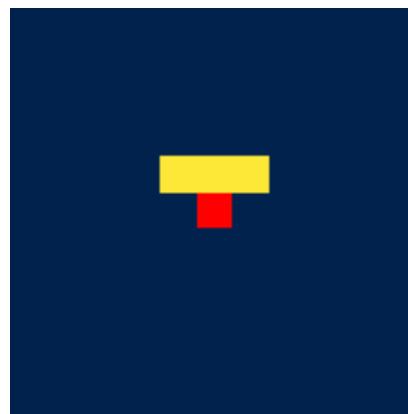
Next, let's take a look at the vertical convolution:

```
[9]: vert_conv = VerticalStackConvolution(c_in=1, c_out=1, kernel_size=3, mask_center=True)
vert_conv.conv.weight.data.fill_(1)
vert_conv.conv.bias.data.fill_(0)
vert_img = vert_conv(inp_img)
show_center_recep_field(inp_img, vert_img)
```

Weighted receptive field



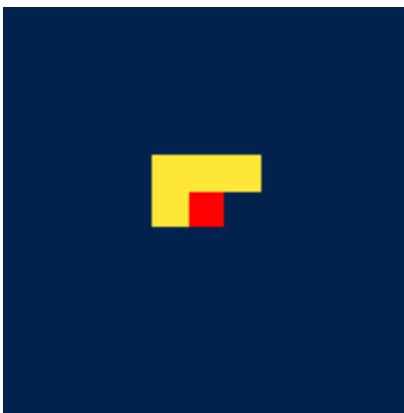
Binary receptive field



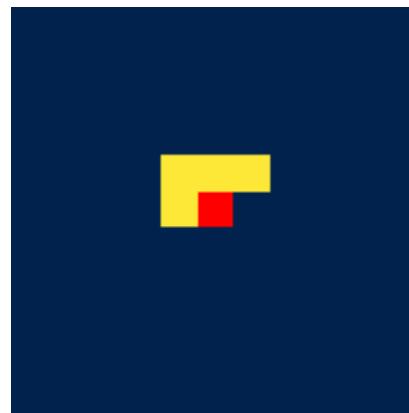
The vertical convolution takes all pixels above into account. Combining these two, we get the L-shaped receptive field of the original masked convolution:

```
[10]: horiz_img = vert_img + horiz_img
show_center_recep_field(inp_img, horiz_img)
```

Weighted receptive field



Binary receptive field



If we stack multiple horizontal and vertical convolutions, we need to take two aspects into account:

1. The center should not be masked anymore for the following convolutions as the features at the pixel's position are already independent of its actual value. If it is hard to imagine why we can do this, just change the value below to `mask_center=True` and see what happens.
2. The vertical convolution is not allowed to work on features from the horizontal convolution. In the feature map of the horizontal convolutions, a pixel contains information about all of the “true” pixels on the left. If we apply a vertical convolution which also uses features from the right, we effectively expand our receptive field to the true input which we want to prevent. Thus, the feature maps can only be merged for the horizontal convolution.

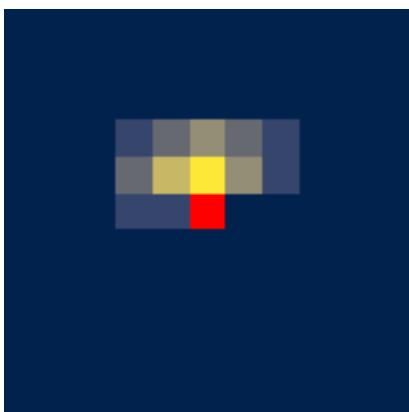
Using this, we can stack the convolutions in the following way. We have two feature streams: one for the vertical stack, and one for the horizontal stack. The horizontal convolutions can operate on the joint features of the previous horizontals and vertical convolutions, while the vertical stack only takes its own previous features as input. For a quick implementation, we can therefore sum the horizontal and vertical output features at each layer, and use those as final output features to calculate the loss on. An implementation of 4 consecutive layers is shown below. Note that we reuse the features from the other convolutions with `mask_center=True` from above.

```
[11]: # Initialize convolutions with equal weight to all input pixels
horiz_conv = HorizontalStackConvolution(c_in=1, c_out=1, kernel_size=3, mask_
    ↴center=False)
horiz_conv.conv.weight.data.fill_(1)
horiz_conv.conv.bias.data.fill_(0)
vert_conv = VerticalStackConvolution(c_in=1, c_out=1, kernel_size=3, mask_center=False)
vert_conv.conv.weight.data.fill_(1)
vert_conv.conv.bias.data.fill_(0)

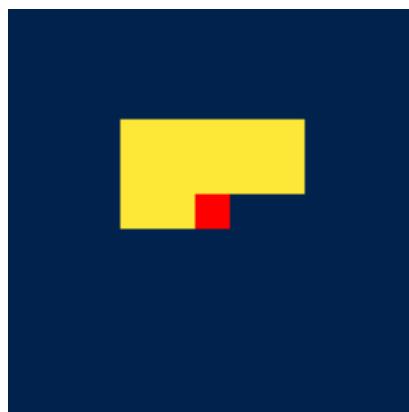
# We reuse our convolutions for the 4 layers here. Note that in a standard network,
# we don't do that, and instead learn 4 separate convolution. As this cell is only for
# visualization purposes, we reuse the convolutions for all layers.
for l_idx in range(4):
    vert_img = vert_conv(vert_img)
    horiz_img = horiz_conv(horiz_img) + vert_img
    print(f"Layer {l_idx+2}")
    show_center_recep_field(inp_img, horiz_img)
```

Layer 2

Weighted receptive field

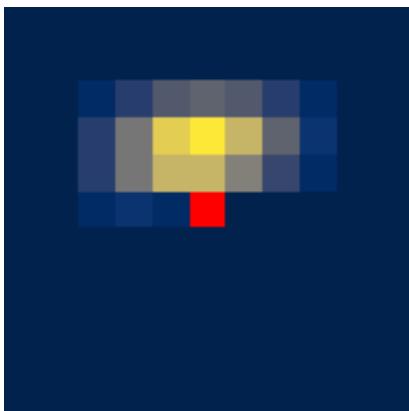


Binary receptive field

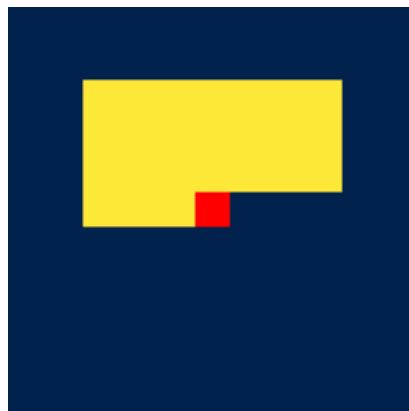


Layer 3

Weighted receptive field

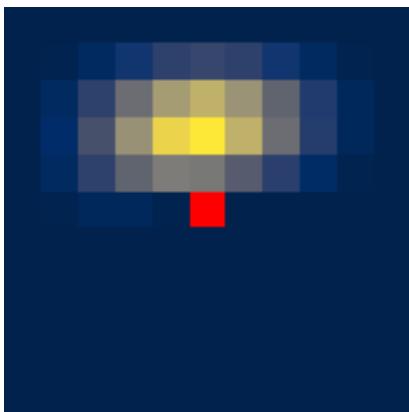


Binary receptive field

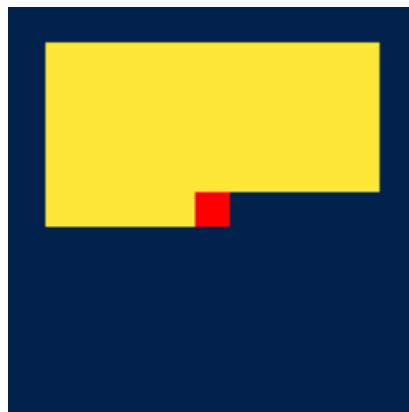


Layer 4

Weighted receptive field

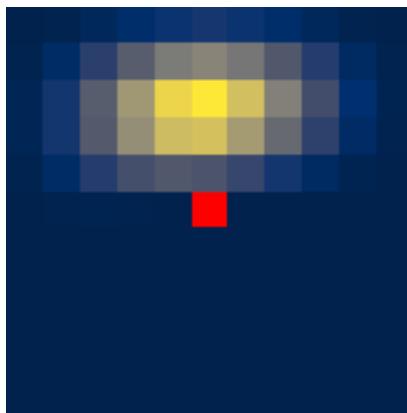


Binary receptive field



Layer 5

Weighted receptive field



Binary receptive field

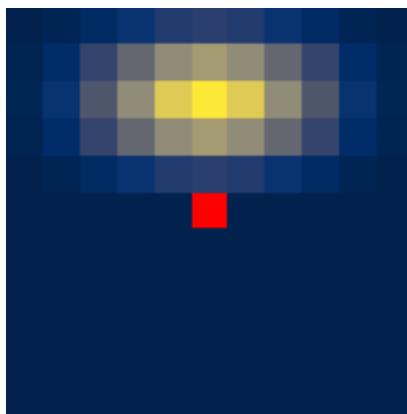


The receptive field above it visualized for the horizontal stack, which includes the features of the vertical convolutions. It grows over layers without any blind spot as we had before. The difference between “weighted” and “binary” receptive field is that for the latter, we check whether there are any gradients flowing back to this pixel. This indicates that the center pixel indeed can use information from this pixel. Nevertheless, due to the convolution weights, some pixels have a stronger effect on the prediction than others. This is visualized in the weighted receptive field by plotting the gradient magnitude for each pixel instead of a binary yes/no.

Another receptive field we can check is the one for the vertical stack as the one above is for the horizontal stack. Let's visualize it below:

```
[12]: show_center_recep_field(inp_img, vert_img)
```

Weighted receptive field



Binary receptive field



As we have discussed before, the vertical stack only looks at pixels above the one we want to predict. Hence, we can validate that our implementation works as we initially expected it to. As a final step, let's clean up the computation graph we still had kept in memory for the visualization of the receptive field:

```
[13]: del inp_img, horiz_conv, vert_conv
```

4.15.2 Gated PixelCNN

In the next step, we will use the masked convolutions to build a full autoregressive model, called Gated PixelCNN. The difference between the original PixelCNN and Gated PixelCNN is the use of separate horizontal and vertical stacks. However, in literature, you often see that people refer to the Gated PixelCNN simply as “PixelCNN”. Hence, in the following, if we say “PixelCNN”, we usually mean the gated version. What “Gated” refers to in the model name is explained next.

Gated Convolutions

For visualizing the receptive field, we assumed a very simplified stack of vertical and horizontal convolutions. Obviously, there are more sophisticated ways of doing it, and PixelCNN uses gated convolutions for this. Specifically, the Gated Convolution block in PixelCNN looks as follows (figure credit - [Aaron van den Oord et al.](#)):

The left path is the vertical stack (the $N \times N$ convolution is masked correspondingly), and the right path is the horizontal stack. Gated convolutions are implemented by having a twice as large output channel size, and combine them by a element-wise multiplication of tanh and a sigmoid. For a linear layer, we can express a gated activation unit as follows:

$$\mathbf{y} = \tanh(\mathbf{W}_f \mathbf{x}) \odot \sigma(\mathbf{W}_g \mathbf{x})$$

For simplicity, biases have been neglected and the linear layer split into two part, \mathbf{W}_f and \mathbf{W}_g . This concept resembles the input and modulation gate in an LSTM, and has been used in many other architectures as well. The main motivation behind this gated activation is that it might allow to model more complex interactions and simplifies learning. But as in any other architecture, this is mostly a design choice and can be considered a hyperparameters.

Besides the gated convolutions, we also see that the horizontal stack uses a residual connection while the vertical stack does not. This is because we use the output of the horizontal stack for prediction. Each convolution in the vertical stack also receives a strong gradient signal as it is only two 1×1 convolutions away from the residual connection, and does not require another residual connection to all its earlier layers.

The implementation in PyTorch is fairly straight forward for this block, because the visualization above gives us a computation graph to follow:

```
[14]: class GatedMaskedConv(nn.Module):

    def __init__(self, c_in, **kwargs):
        """
        Gated Convolution block implemented the computation graph shown above.
        """
        super().__init__()
        self.conv_vert = VerticalStackConvolution(c_in, c_out=2*c_in, **kwargs)
        self.conv_horiz = HorizontalStackConvolution(c_in, c_out=2*c_in, **kwargs)
        self.conv_vert_to_horiz = nn.Conv2d(2*c_in, 2*c_in, kernel_size=1, padding=0)
        self.conv_horiz_1x1 = nn.Conv2d(c_in, c_in, kernel_size=1, padding=0)

    def forward(self, v_stack, h_stack):
        # Vertical stack (left)
        v_stack_feat = self.conv_vert(v_stack)
        v_val, v_gate = v_stack_feat.chunk(2, dim=1)
        v_stack_out = torch.tanh(v_val) * torch.sigmoid(v_gate)

        # Horizontal stack (right)
```

(continues on next page)

(continued from previous page)

```

h_stack_feat = self.conv_horiz(h_stack)
h_stack_feat = h_stack_feat + self.conv_vert_to_horiz(v_stack_feat)
h_val, h_gate = h_stack_feat.chunk(2, dim=1)
h_stack_feat = torch.tanh(h_val) * torch.sigmoid(h_gate)
h_stack_out = self.conv_horiz_1x1(h_stack_feat)
h_stack_out = h_stack_out + h_stack

return v_stack_out, h_stack_out

```

Building the model

Using the gated convolutions, we can now build our PixelCNN model. The architecture consists of multiple stacked GatedMaskedConv blocks, where we add an additional dilation factor to a few convolutions. This is used to increase the receptive field of the model and allows to take a larger context into account during generation. As a reminder, dilation on a convolution works as follows (figure credit - [Vincent Dumoulin and Francesco Visin](#)):

Note that the smaller output size is only because the animation assumes no padding. In our implementation, we will pad the input image correspondingly. Alternatively to dilated convolutions, we could downsample the input and use an encoder-decoder architecture as in PixelCNN++ [3]. This is especially beneficial if we want to build a very deep autoregressive model. Nonetheless, as we seek to train a reasonably small model, dilated convolutions are the more efficient option to use here.

Below, we implement the PixelCNN model as a PyTorch Lightning module. Besides the stack of gated convolutions, we also have the initial horizontal and vertical convolutions which mask the center pixel, and a final 1×1 convolution which maps the output features to class predictions. To determine the likelihood of a batch of images, we first create our initial features using the masked horizontal and vertical input convolution. Next, we forward the features through the stack of gated convolutions. Finally, we take the output features of the horizontal stack, and apply the 1×1 convolution for classification. We use the bits per dimension metric for the likelihood, similarly to Tutorial 11 and assignment 3.

```
[15]: class PixelCNN(pl.LightningModule):

    def __init__(self, c_in, c_hidden):
        super().__init__()
        self.save_hyperparameters()

        # Initial convolutions skipping the center pixel
        self.conv.vstack = VerticalStackConvolution(c_in, c_hidden, mask_center=True)
        self.conv.hstack = HorizontalStackConvolution(c_in, c_hidden, mask_center=True)
        # Convolution block of PixelCNN. We use dilation instead of downscaling
        self.conv_layers = nn.ModuleList([
            GatedMaskedConv(c_hidden),
            GatedMaskedConv(c_hidden, dilation=2),
            GatedMaskedConv(c_hidden),
            GatedMaskedConv(c_hidden, dilation=4),
            GatedMaskedConv(c_hidden),
            GatedMaskedConv(c_hidden, dilation=2),
            GatedMaskedConv(c_hidden)
        ])
        # Output classification convolution (1x1)
        self.conv_out = nn.Conv2d(c_hidden, c_in * 256, kernel_size=1, padding=0)
```

(continues on next page)

(continued from previous page)

```

    self.example_input_array = train_set[0][0][None]

def forward(self, x):
    """
    Forward image through model and return logits for each pixel.
    Inputs:
        x - Image tensor with integer values between 0 and 255.
    """
    # Scale input from 0 to 255 back to -1 to 1
    x = (x.float() / 255.0) * 2 - 1

    # Initial convolutions
    v_stack = self.conv_vstack(x)
    h_stack = self.conv_hstack(x)
    # Gated Convolutions
    for layer in self.conv_layers:
        v_stack, h_stack = layer(v_stack, h_stack)
    # 1x1 classification convolution
    # Apply ELU before 1x1 convolution for non-linearity on residual connection
    out = self.conv_out(F.elu(h_stack))

    # Output dimensions: [Batch, Classes, Channels, Height, Width]
    out = out.reshape(out.shape[0], 256, out.shape[1]//256, out.shape[2], out.
shape[3])
    return out

def calc_likelihood(self, x):
    # Forward pass with bpd likelihood calculation
    pred = self.forward(x)
    nll = F.cross_entropy(pred, x, reduction='none')
    bpd = nll.mean(dim=[1, 2, 3]) * np.log2(np.exp(1))
    return bpd.mean()

@torch.no_grad()
def sample(self, img_shape, img=None):
    """
    Sampling function for the autoregressive model.
    Inputs:
        img_shape - Shape of the image to generate (B, C, H, W)
        img (optional) - If given, this tensor will be used as
                        a starting image. The pixels to fill
                        should be -1 in the input tensor.
    """
    # Create empty image
    if img is None:
        img = torch.zeros(img_shape, dtype=torch.long).to(device) - 1
    # Generation loop
    for h in tqdm(range(img_shape[2]), leave=False):
        for w in range(img_shape[3]):
            for c in range(img_shape[1]):
                # Skip if not to be filled (-1)

```

(continues on next page)

(continued from previous page)

```

if (img[:,c,h,w] != -1).all().item():
    continue
# For efficiency, we only have to input the upper part of the image
# as all other parts will be skipped by the masked convolutions.
→ anyways
pred = self.forward(img[:, :, :h+1, :])
probs = F.softmax(pred[:, :, c, h, w], dim=-1)
img[:, c, h, w] = torch.multinomial(probs, num_samples=1).squeeze(dim=-1)
return img

def configure_optimizers(self):
    optimizer = optim.Adam(self.parameters(), lr=1e-3)
    scheduler = optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.99)
    return [optimizer], [scheduler]

def training_step(self, batch, batch_idx):
    loss = self.calc_likelihood(batch[0])
    self.log('train_bpd', loss)
    return loss

def validation_step(self, batch, batch_idx):
    loss = self.calc_likelihood(batch[0])
    self.log('val_bpd', loss)

def test_step(self, batch, batch_idx):
    loss = self.calc_likelihood(batch[0])
    self.log('test_bpd', loss)

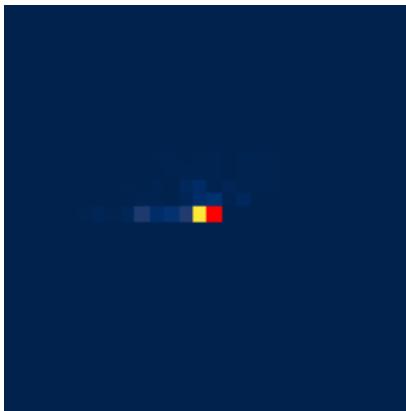
```

To sample from the autoregressive model, we need to iterate over all dimensions of the input. We start with an empty image, and fill the pixels one by one, starting from the upper left corner. Note that as for predicting x_i , all pixels below it have no influence on the prediction. Hence, we can cut the image in height without changing the prediction while increasing efficiency. Nevertheless, all the loops in the sampling function already show that it will take us quite some time to sample. A lot of computation could be reused across loop iterations as those the features on the already predicted pixels will not change over iterations. Nevertheless, this takes quite some effort to implement, and is often not done in implementations because in the end, autoregressive sampling remains sequential and slow. Hence, we settle with the default implementation here.

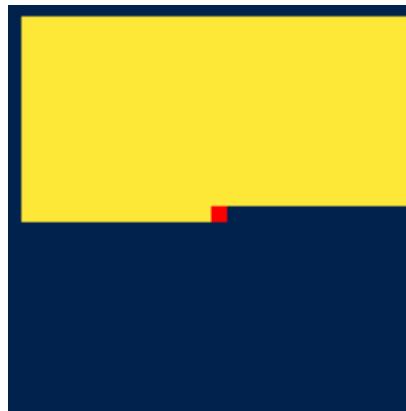
Before training the model, we can check the full receptive field of the model on an MNIST image of size 28×28 :

```
[16]: test_model = PixelCNN(c_in=1, c_hidden=64)
inp = torch.zeros(1,1,28,28)
inp.requires_grad_()
out = test_model(inp)
show_center_recep_field(inp, out.squeeze(dim=2))
del inp, out, test_model
```

Weighted receptive field



Binary receptive field



The visualization shows that for predicting any pixel, we can take almost half of the image into account. However, keep in mind that this is the “theoretical” receptive field and not necessarily the [effective receptive field](#), which is usually much smaller. For a stronger model, we should therefore try to increase the receptive field even further. Especially, for the pixel on the bottom right, the very last pixel, we would be allowed to take into account the whole image. However, our current receptive field only spans across 1/4 of the image. An encoder-decoder architecture can help with this, but it also shows that we require a much deeper, more complex network in autoregressive models than in VAEs or energy-based models.

Training loop

To train the model, we again can rely on PyTorch Lightning and write a function below for loading the pretrained model if it exists. To reduce the computational cost, we have saved the validation and test score in the checkpoint already:

```
[17]: def train_model(**kwargs):
    # Create a PyTorch Lightning trainer with the generation callback
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "PixelCNN"),
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=150,
                          callbacks=[ModelCheckpoint(save_weights_only=True, mode="min",  
monitor="val_bpd"),
                                     LearningRateMonitor("epoch")])
    result = None
    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "PixelCNN.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = PixelCNN.load_from_checkpoint(pretrained_filename)
        ckpt = torch.load(pretrained_filename, map_location=device)
        result = ckpt.get("result", None)
    else:
        model = PixelCNN(**kwargs)
        trainer.fit(model, train_loader, val_loader)
        model = model.to(device)

    if result is None:
```

(continues on next page)

(continued from previous page)

```
# Test best model on validation and test set
val_result = trainer.test(model, val_loader, verbose=False)
test_result = trainer.test(model, test_loader, verbose=False)
result = {"test": test_result, "val": val_result}
return model, result
```

Training the model is time consuming and we recommend using the provided pre-trained model for going through this notebook. However, feel free to play around with the hyperparameter like number of layers etc. if you want to get a feeling for those.

When calling the training function with a pre-trained model, we automatically load it and print its test performance:

```
[18]: model, result = train_model(c_in=1, c_hidden=64)
test_res = result["test"][@]
print("Test bits per dimension: %4.3fbpd" % (test_res["test_loss"] if "test_loss" in
                                              test_res else test_res["test_bpd"]))
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
Found pretrained model, loading...
Test bits per dimension: 0.808bpd
```

With a test performance of 0.809bpd, the PixelCNN significantly outperforms the normalizing flows we have seen in Tutorial 11. Considering image modeling as an autoregressive problem simplifies the learning process as predicting one pixel given the ground truth of all others is much easier than predicting all pixels at once. In addition, PixelCNN can explicitly predict the pixel values by a discrete softmax while Normalizing Flows have to learn transformations in continuous latent space. These two aspects allow the PixelCNN to achieve a notably better performance.

To fully compare the models, let's also measure the number of parameters of the PixelCNN:

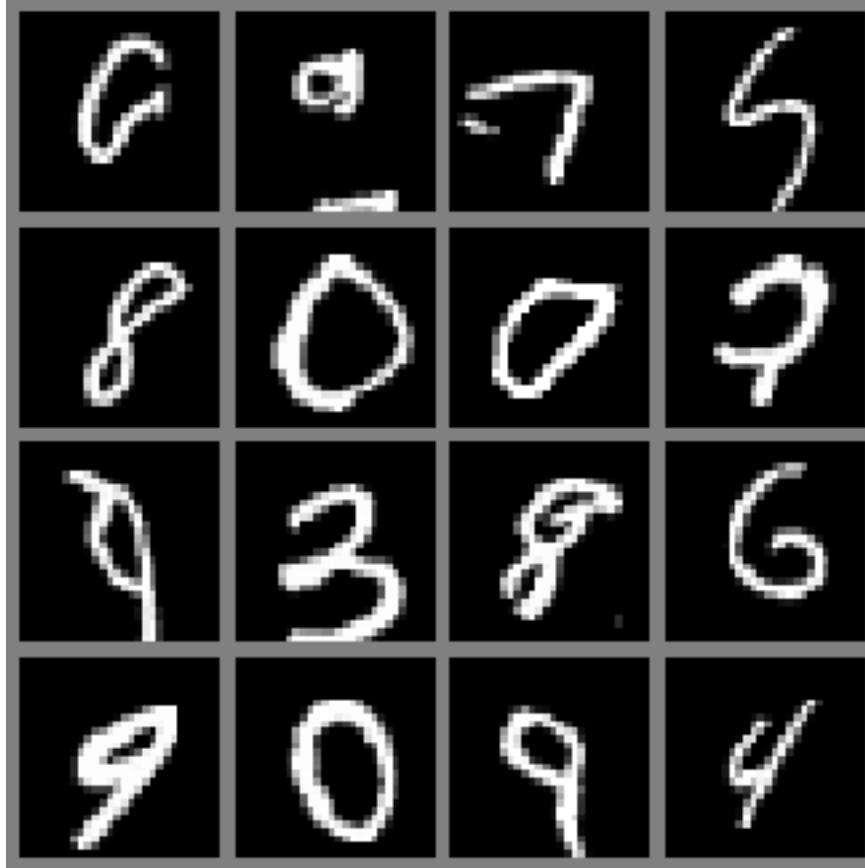
```
[19]: num_params = sum([np.prod(param.shape) for param in model.parameters()])
print("Number of parameters: {}".format(num_params))
Number of parameters: 852,160
```

Compared to the multi-scale normalizing flows, the PixelCNN has considerably less parameters. Of course, the number of parameters depend on our hyperparameter choices. Nevertheless, in general, it can be said that autoregressive models require considerably less parameters than normalizing flows to reach good performance, based on the reasons stated above. Still, autoregressive models are much slower in sampling than normalizing flows, which limits their possible applications.

4.15.3 Sampling

One way of qualitatively analysing generative models is by looking at the actual samples. Let's therefore use our sampling function to generate a few digits:

```
[20]: pl.seed_everything(1)
samples = model.sample(img_shape=(16, 1, 28, 28))
show_imgs(samples.cpu())
HBox(children=(FloatProgress(value=0.0, max=28.0), HTML(value='')))
```



Most of the samples can be identified as digits, and overall we achieve a better quality than we had in normalizing flows. This goes along with the lower likelihood we achieved with autoregressive models. Nevertheless, we also see that there is still place for improvement as a considerable amount of samples cannot be identified (for example the first row). Deeper autoregressive models are expected to achieve better quality, as they can take more context into account for generating the pixels.

Note that on Google Colab, you might see different results, specifically with a white line at the top. After some debugging, it seemed that the difference occurs inside the dilated convolution, as it gives different results for different batch sizes. However, it is hard to debug this further as it might be a bug of the installed PyTorch version on Google Colab.

The trained model itself is not restricted to any specific image size. However, what happens if we actually sample a larger image than we had seen in our training dataset? Let's try below to sample images of size 64×64 instead of 28×28 :

```
[21]: pl.seed_everything(1)
samples = model.sample(img_shape=(8, 1, 64, 64))
show_imgs(samples.cpu())
HBox(children=(FloatProgress(value=0.0, max=64.0), HTML(value='')))
```



The larger images show that changing the size of the image during testing confuses the model and generates abstract figures (you can sometimes spot a digit in the upper left corner). In addition, sampling for images of 64x64 pixels take more than a minute on a GPU. Clearly, autoregressive models cannot be scaled to large images without changing the sampling procedure such as with [forecasting](#). Our implementation is also not the most efficient as many computations can be stored and reused throughout the sampling process. Nevertheless, the sampling procedure stays sequential which is inherently slower than parallel generation like done in normalizing flows.

Autocompletion

One common application done with autoregressive models is auto-completing an image. As autoregressive models predict pixels one by one, we can set the first N pixels to predefined values and check how the model completes the image. For implementing this, we just need to skip the iterations in the sampling loop that already have a value unequal -1. See above in our PyTorch Lightning module for the specific implementation. In the cell below, we randomly take three images from the training set, mask about the lower half of the image, and let the model autocomplete it. To see the diversity of samples, we do this 12 times for each image:

```
[22]: def autocomplete_image(img):
    # Remove lower half of the image
    img_init = img.clone()
    img_init[:,10:,:,:] = -1
    print("Original image and input image to sampling:")
    show_imgs([img,img_init])
    # Generate 12 example completions
    img_init = img_init.unsqueeze(dim=0).expand(12,-1,-1,-1).to(device)
    pl.seed_everything(1)
    img_generated = model.sample(img_init.shape, img_init)
    print("Autocompletion samples:")
    show_imgs(img_generated)

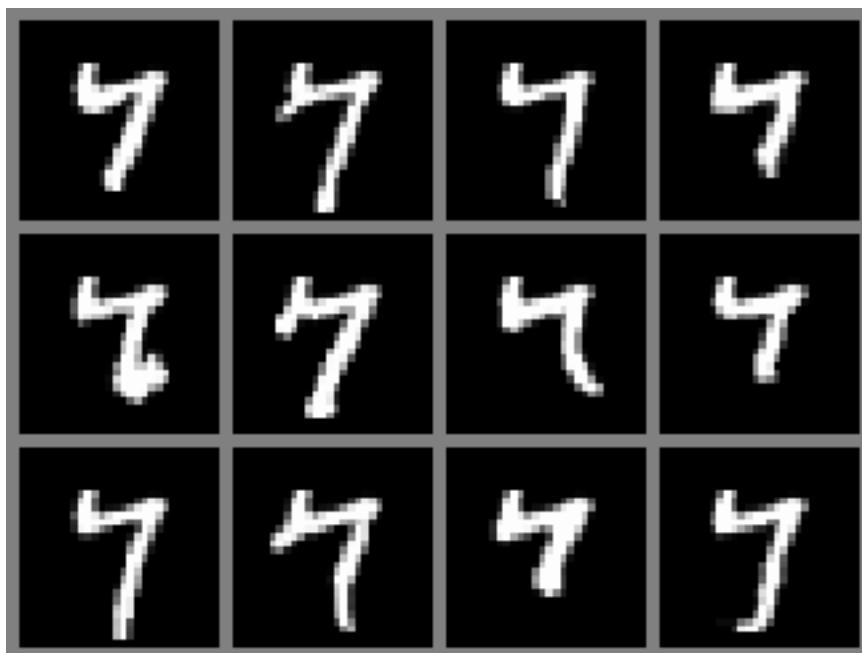
for i in range(1,4):
    img = train_set[i][0]
    autocomplete_image(img)

Original image and input image to sampling:
```



```
HBox(children=(FloatProgress(value=0.0, max=28.0), HTML(value='')))
```

Autocompletion samples:

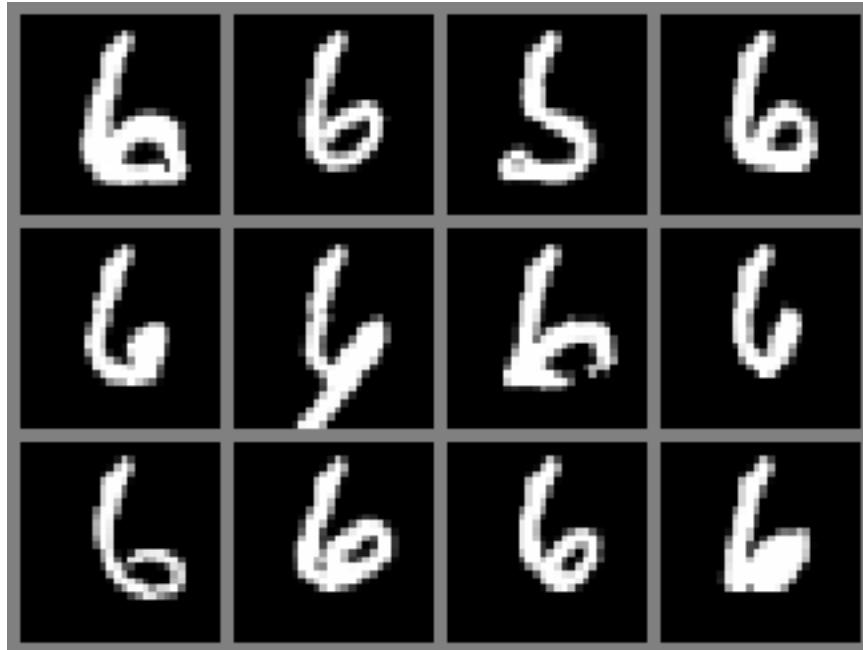


Original image and input image to sampling:



```
HBox(children=(FloatProgress(value=0.0, max=28.0), HTML(value='')))
```

Autocompletion samples:

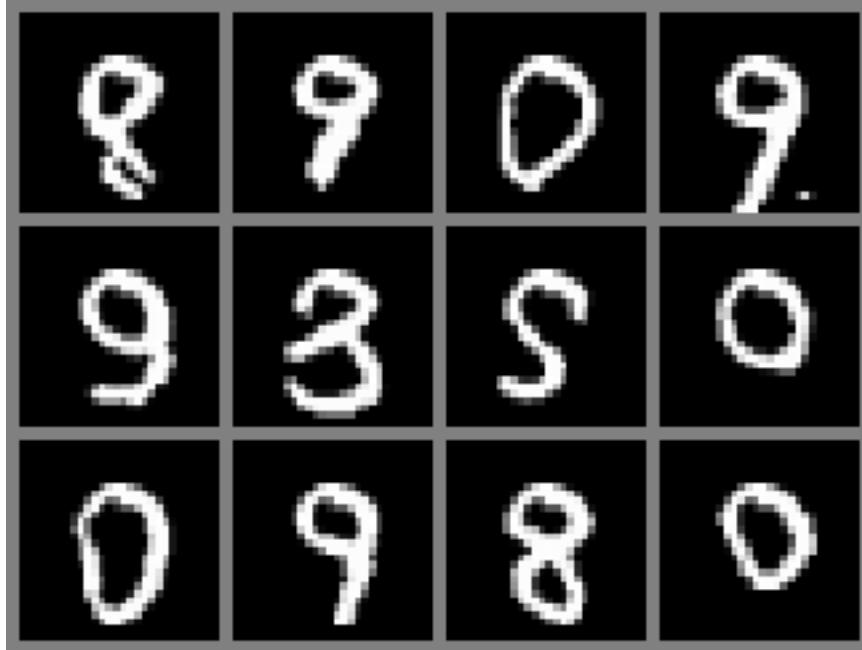


Original image and input image to sampling:



```
HBox(children=(FloatProgress(value=0.0, max=28.0), HTML(value='')))
```

Autocompletion samples:



For the first two digits (7 and 6), we see that the 12 samples all result in a shape which resemble the original digit. Nevertheless, there are some style difference in writing the 7, and some deformed sixes in the samples. When auto-completing the 9 below, we see that the model can fit multiple digits to it. We obtain diverse samples from 0, 3, 8 and 9. This shows that despite having no latent space, we can still obtain diverse samples from an autoregressive model.

Visualization of the predictive distribution (softmax)

Autoregressive models use a softmax over 256 values to predict the next pixel. This gives the model a large flexibility as the probabilities for each pixel value can be learned independently if necessary. However, the values are actually not independent because the values 32 and 33 are much closer than 32 and 255. In the following, we visualize the softmax distribution that the model predicts to gain insights how it has learned the relationships of close-by pixels.

To do this, we first run the model on a batch of images and store the output softmax distributions:

```
[23]: det_loader = data.DataLoader(train_set, batch_size=128, shuffle=False, drop_last=False)
imgs, _ = next(iter(det_loader))
imgs = imgs.to(device)
with torch.no_grad():
    out = model(imgs)
    out = F.softmax(out, dim=1)
    mean_out = out.mean(dim=[0, 2, 3, 4]).cpu().numpy()
    out = out.cpu().numpy()
```

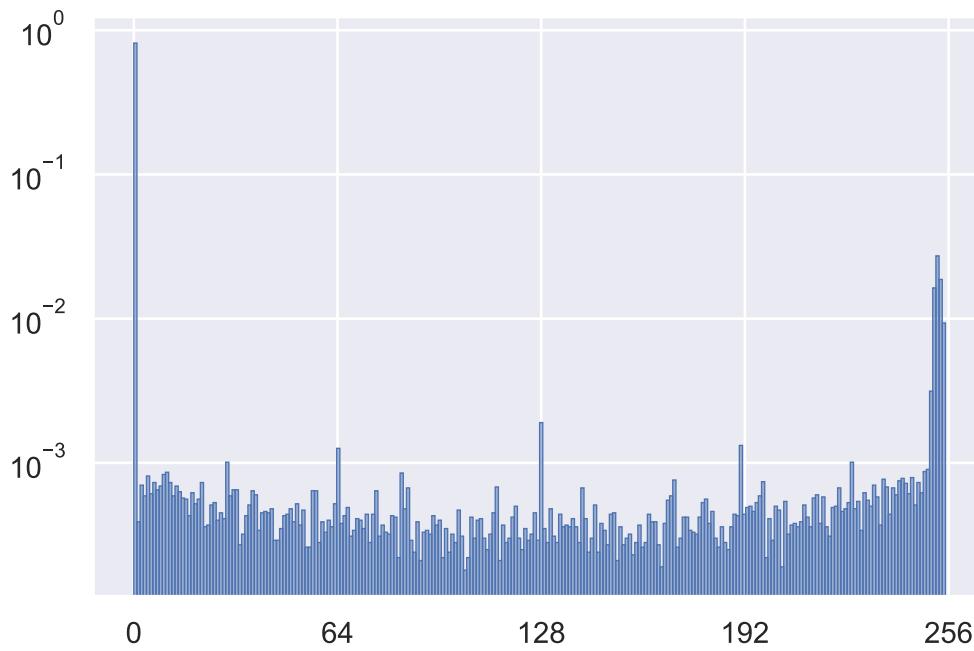
Before diving into the model, let's visualize the distribution of the pixel values in the whole dataset:

```
[24]: sns.set()
plot_args = {"color": to_rgb("C0")+(0.5,), "edgecolor": "C0", "linewidth": 0.5, "width": 1.0}
plt.hist(imgs.view(-1).cpu().numpy(), bins=256, density=True, **plot_args)
plt.yscale("log")
```

(continues on next page)

(continued from previous page)

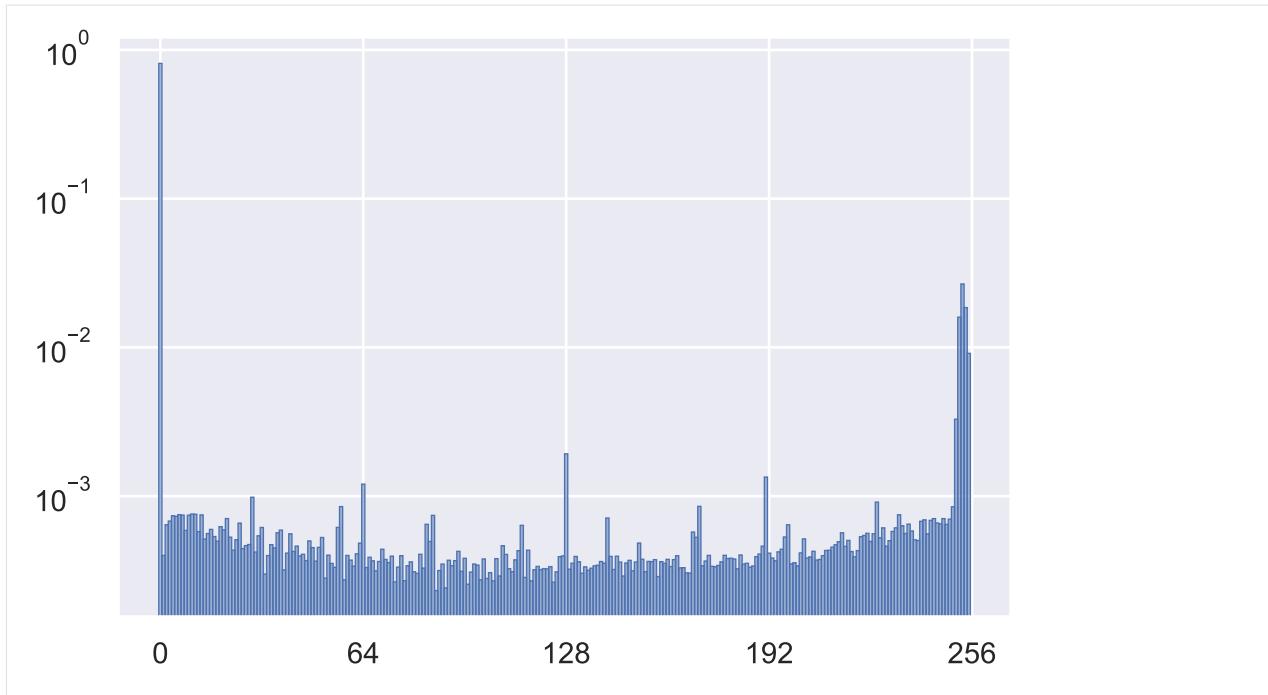
```
plt.xticks([0,64,128,192,256])
plt.show()
plt.close()
```



As we would expect from the seen images, the pixel value 0 (black) is the dominant value, followed by a batch of values between 250 and 255. Note that we use a log scale on the y-axis due to the big imbalance in the dataset. Interestingly, the pixel values 64, 128 and 191 also stand out which is likely due to the quantization used during the creation of the dataset. For RGB images, we would also see two peaks around 0 and 255, but the values in between would be much more frequent than in MNIST (see Figure 1 in the [PixelCNN++](#) for a visualization on CIFAR10).

Next, we can visualize the distribution our model predicts (in average):

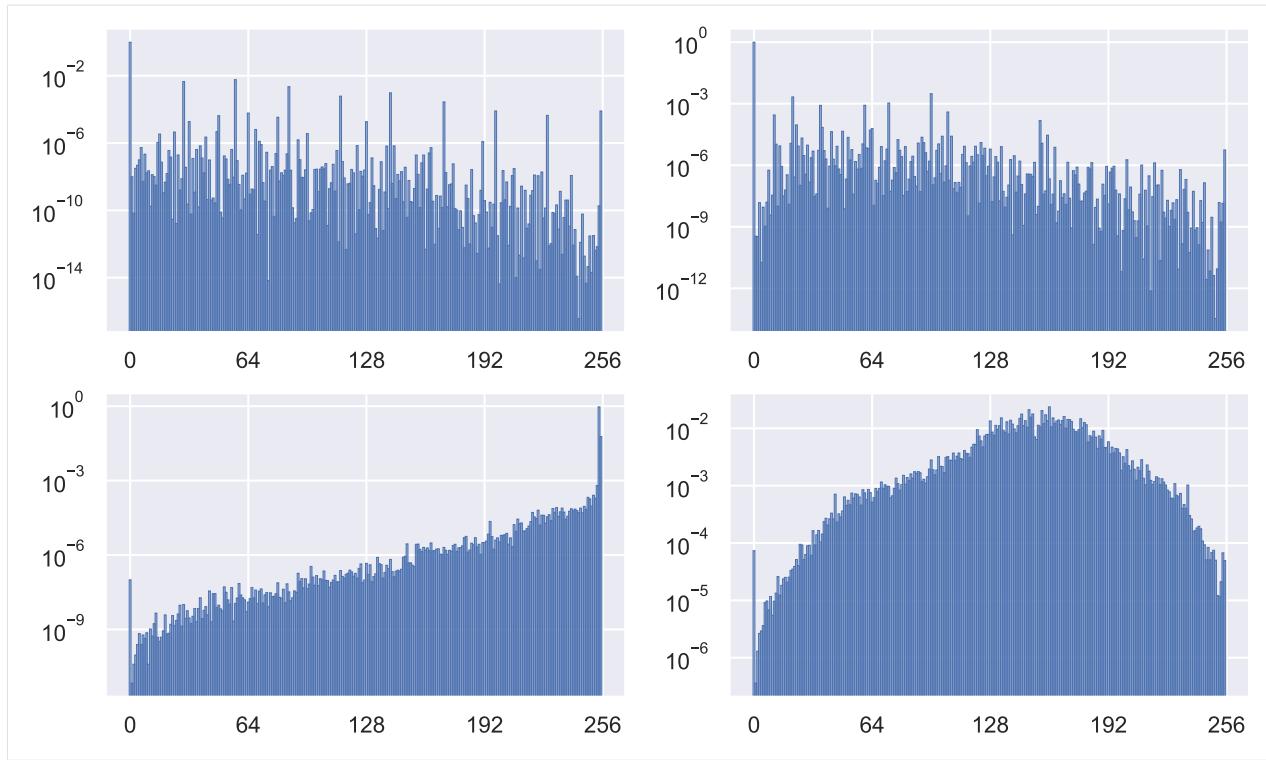
```
[25]: plt.bar(np.arange(mean_out.shape[0]), mean_out, **plot_args)
plt.yscale("log")
plt.xticks([0,64,128,192,256])
plt.show()
plt.close()
```



This distribution is very close to the actual dataset distribution. This is in general a good sign, but we can see a slightly smoother histogram than above.

Finally, to take a closer look at learned value relations, we can visualize the distribution for individual pixel predictions to get a better intuition. For this, we pick 4 random images and pixels, and visualize their distribution below:

```
[26]: fig, ax = plt.subplots(2,2, figsize=(10,6))
for i in range(4):
    ax_sub = ax[i//2][i%2]
    ax_sub.bar(np.arange(out.shape[1], dtype=np.int32), out[i+4,:,0,14,14], **plot_args)
    ax_sub.set_yscale("log")
    ax_sub.set_xticks([0,64,128,192,256])
plt.show()
plt.close()
```



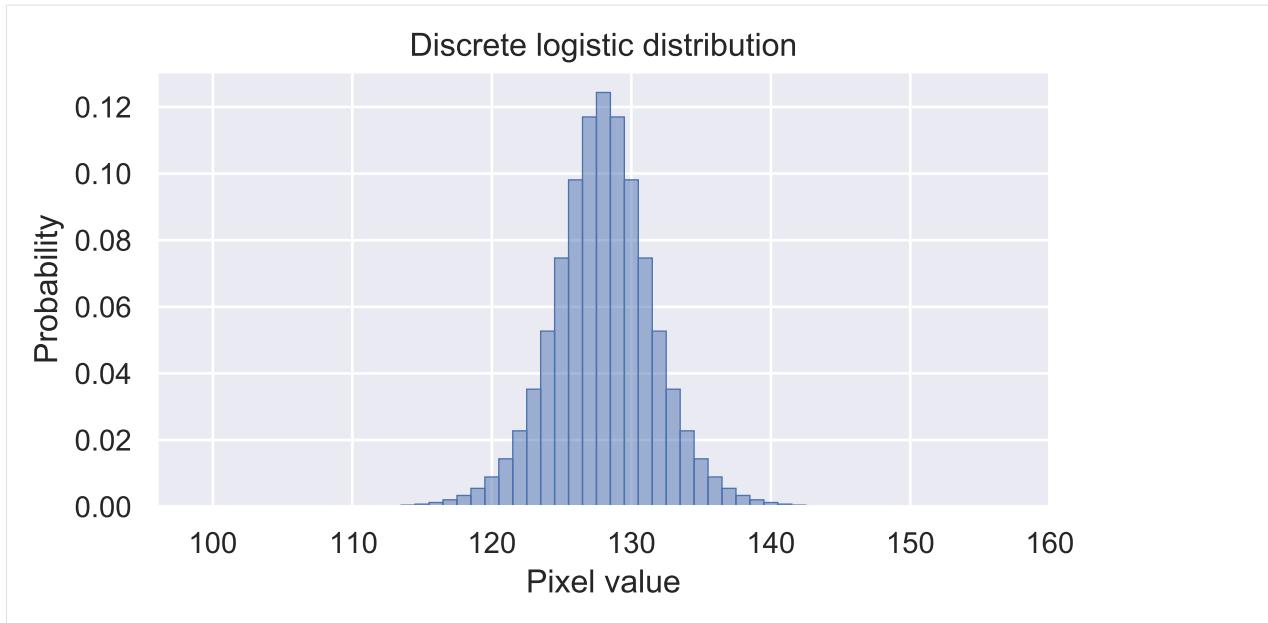
Overall we see a very diverse set of distributions, with a usual peak for 0 and close to 1. However, the distributions in the first row show a potentially undesirable behavior. For instance, the value 242 has a 1000x lower likelihood than 243 although they are extremely close and can often not be distinguished. This shows that the model might have not generalized well over pixel values. The better solution to this problem is to use discrete logistics mixtures instead of a softmax distribution. A discrete logistic distribution can be imagined as discretized, binned Gaussians. Using a mixture of discrete logistics instead of a softmax introduces an inductive bias to the model to assign close-by values similar likelihoods. We can visualize a discrete logistic below:

```
[27]: mu = torch.Tensor([128])
sigma = torch.Tensor([2.0])

def discrete_logistic(x, mu, sigma):
    return torch.sigmoid((x+0.5-mu)/sigma) - torch.sigmoid((x-0.5-mu)/sigma)

x = torch.arange(256)
p = discrete_logistic(x, mu, sigma)

# Visualization
plt.figure(figsize=(6,3))
plt.bar(x.numpy(), p.numpy(), **plot_args)
plt.xlim(96,160)
plt.title("Discrete logistic distribution")
plt.xlabel("Pixel value")
plt.ylabel("Probability")
plt.show()
plt.close()
```



Instead of the softmax, the model would output mean and standard deviations for the K logistics we use in the mixture. This is one of the improvements in autoregressive models that PixelCNN++ [3] has introduced compared to the original PixelCNN.

4.15.4 Conclusion

In this tutorial, we have looked at autoregressive image modeling, and implemented the PixelCNN architecture. With the usage of masked convolutions, we are able to apply a convolutional network in which a pixel is only influenced by all its predecessors. Separating the masked convolution into a horizontal and vertical stack allowed us to remove the known blind spot on the right upper row of a pixel. In experiments, autoregressive models outperformed normalizing flows in terms of bits per dimension, but are much slower to sample from. Improvements, that we have not implemented ourselves here, are discrete logistic mixtures, a downsampling architecture, and changing the pixel order in a diagonal fashion (see PixelSNAIL). Overall, autoregressive models are another, strong family of generative models, which however are mostly used in sequence tasks because of their linear scaling in sampling time than quadratic as on images.

4.15.5 References

- [1] van den Oord, A., et al. “Pixel Recurrent Neural Networks.” arXiv preprint arXiv:1601.06759 (2016). [Link](#)
- [2] van den Oord, A., et al. “Conditional Image Generation with PixelCNN Decoders.” In Advances in Neural Information Processing Systems 29, pp. 4790–4798 (2016). [Link](#)
- [3] Salimans, Tim, et al. “PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications.” arXiv preprint arXiv:1701.05517 (2017). [Link](#)

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.16 Tutorial 15: Vision Transformers

Filled notebook:

Pre-trained models:

Recordings:

JAX+Flax version:

Author: Phillip Lippe

Note: Interested in JAX? Check out our [JAX+Flax version](#) of this tutorial!

In this tutorial, we will take a closer look at a recent new trend: Transformers for Computer Vision. Since [Alexey Dosovitskiy et al.](#) successfully applied a Transformer on a variety of image recognition benchmarks, there have been an incredible amount of follow-up works showing that CNNs might not be optimal architecture for Computer Vision anymore. But how do Vision Transformers work exactly, and what benefits and drawbacks do they offer in contrast to CNNs? We will answer these questions by implementing a Vision Transformer ourselves and train it on the popular, small dataset CIFAR10. We will compare these results to the convolutional architectures of [Tutorial 5](#).

If you are not familiar with Transformers yet, take a look at [Tutorial 6](#) where we discuss the fundamentals of Multi-Head Attention and Transformers. As in many previous tutorials, we will use [PyTorch Lightning](#) again (introduced in [Tutorial 5](#)). Let's start with importing our standard set of libraries.

```
[1]: ## Standard libraries
import os
import numpy as np
import random
import math
import json
from functools import partial
from PIL import Image

## Imports for plotting
import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## tqdm for loading bars
from tqdm.notebook import tqdm
```

(continues on next page)

(continued from previous page)

```

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim

## Torchvision
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms

# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Import tensorboard
%load_ext tensorboard

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial15"

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", device)

Global seed set to 42
Device: cuda:0

```

We provide a pre-trained Vision Transformer which we download in the next cell. However, Vision Transformers can be relatively quickly trained on CIFAR10 with an overall training time of less than an hour on an NVIDIA TitanRTX. Feel free to experiment with training your own Transformer once you went through the whole notebook.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/"
# Files to download
```

(continues on next page)

(continued from previous page)

```

pretrained_files = ["tutorial15/ViT.ckpt", "tutorial15/tensorboards/ViT/events.out.
˓→tfevents.ViT",
                    "tutorial5/tensorboards/ResNet/events.out.tfevents.resnet"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name.split("/", 1)[1])
    if "/" in file_name.split("/", 1)[1]:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive_
˓→folder, or contact the author with the full output including the following error:\n",_
˓→e)

```

We load the CIFAR10 dataset below. We use the same setup of the datasets and data augmentations as for the CNNs in Tutorial 5 to keep a fair comparison. The constants in the `transforms.Normalize` correspond to the values that scale and shift the data to a zero mean and standard deviation of one.

```

[3]: test_transform = transforms.Compose([transforms.ToTensor(),
                                         transforms.Normalize([0.49139968, 0.48215841, 0.
˓→44653091], [0.24703223, 0.24348513, 0.26158784])
                                         ])
# For training, we add some augmentation. Networks are too powerful and would overfit.
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                         transforms.RandomResizedCrop((32, 32), scale=(0.8, 1.
˓→0), ratio=(0.9, 1.1)),
                                         transforms.ToTensor(),
                                         transforms.Normalize([0.49139968, 0.48215841, 0.
˓→44653091], [0.24703223, 0.24348513, 0.26158784])
                                         ])
# Loading the training dataset. We need to split it into a training and validation part
# We need to do a little trick because the validation set should not use the_
˓→augmentation.
train_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=train_transform,
˓→download=True)
val_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=test_transform,
˓→download=True)
pl.seed_everything(42)
train_set, _ = torch.utils.data.random_split(train_dataset, [45000, 5000])
pl.seed_everything(42)
_, val_set = torch.utils.data.random_split(val_dataset, [45000, 5000])

# Loading the test set
test_set = CIFAR10(root=DATASET_PATH, train=False, transform=test_transform,
˓→download=True)

```

(continues on next page)

(continued from previous page)

```
# We define a set of data loaders that we can use for various purposes later.
train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True,
                                pin_memory=True, num_workers=4)
val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False, drop_last=False,
                                num_workers=4)
test_loader = data.DataLoader(test_set, batch_size=128, shuffle=False, drop_last=False,
                                num_workers=4)

# Visualize some examples
NUM_IMAGES = 4
CIFAR_images = torch.stack([val_set[idx][0] for idx in range(NUM_IMAGES)], dim=0)
img_grid = torchvision.utils.make_grid(CIFAR_images, nrow=4, normalize=True, pad_value=0.
                                         )
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("Image examples of the CIFAR10 dataset")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()

Files already downloaded and verified
Files already downloaded and verified

Global seed set to 42
Global seed set to 42

Files already downloaded and verified
```

Image examples of the CIFAR10 dataset



4.16.1 Transformers for image classification

Transformers have been originally proposed to process sets since it is a permutation-equivariant architecture, i.e., producing the same output permuted if the input is permuted. To apply Transformers to sequences, we have simply added a positional encoding to the input feature vectors, and the model learned by itself what to do with it. So, why not do the same thing on images? This is exactly what Alexey Dosovitskiy et al. proposed in their paper “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. Specifically, the Vision Transformer is a model for image classification that views images as sequences of smaller patches. As a preprocessing step, we split an image of, for example, 48×48 pixels into $9 \times 16 \times 16$ patches. Each of those patches is considered to be a “word”/“token” and projected to a feature space. With adding positional encodings and a token for classification on top, we can apply a Transformer as usual to this sequence and start training it for our task. A nice GIF visualization of the architecture is shown below (figure credit - Phil Wang):

We will walk step by step through the Vision Transformer, and implement all parts by ourselves. First, let’s implement the image preprocessing: an image of size $N \times N$ has to be split into $(N/M)^2$ patches of size $M \times M$. These represent the input words to the Transformer.

```
[4]: def img_to_patch(x, patch_size, flatten_channels=True):
    """
    Inputs:
        x - torch.Tensor representing the image of shape [B, C, H, W]
        patch_size - Number of pixels per dimension of the patches (integer)
        flatten_channels - If True, the patches will be returned in a flattened format
                           as a feature vector instead of a image grid.
    """
    B, C, H, W = x.shape
    x = x.reshape(B, C, H//patch_size, patch_size, W//patch_size, patch_size)
    x = x.permute(0, 2, 4, 1, 3, 5) # [B, H', W', C, p_H, p_W]
    x = x.flatten(1, 2)           # [B, H'*W', C, p_H, p_W]
    if flatten_channels:
        x = x.flatten(2, 4)       # [B, H'*W', C*p_H*p_W]
    return x
```

Let’s take a look at how that works for our CIFAR examples above. For our images of size 32×32 , we choose a patch size of 4. Hence, we obtain sequences of 64 patches of size 4×4 . We visualize them below:

```
[5]: img_patches = img_to_patch(CIFAR_images, patch_size=4, flatten_channels=False)

fig, ax = plt.subplots(CIFAR_images.shape[0], 1, figsize=(14,3))
fig.suptitle("Images as input sequences of patches")
for i in range(CIFAR_images.shape[0]):
    img_grid = torchvision.utils.make_grid(img_patches[i], nrow=64, normalize=True, pad_
    value=0.9)
    img_grid = img_grid.permute(1, 2, 0)
    ax[i].imshow(img_grid)
    ax[i].axis('off')
plt.show()
plt.close()
```



Compared to the original images, it is much harder to recognize the objects from those patch lists now. Still, this is the input we provide to the Transformer for classifying the images. The model has to learn itself how it has to combine the patches to recognize the objects. The inductive bias in CNNs that an image is a grid of pixels, is lost in this input format.

After we have looked at the preprocessing, we can now start building the Transformer model. Since we have discussed the fundamentals of Multi-Head Attention in [Tutorial 6](#), we will use the PyTorch module `nn.MultiheadAttention` ([docs](#)) here. Further, we use the Pre-Layer Normalization version of the Transformer blocks proposed by [Ruixin Xiong et al.](#) in 2020. The idea is to apply Layer Normalization not in between residual blocks, but instead as a first layer in the residual blocks. This reorganization of the layers supports better gradient flow and removes the necessity of a warm-up stage. A visualization of the difference between the standard Post-LN and the Pre-LN version is shown below.

The implementation of the Pre-LN attention block looks as follows:

```
[6]: class AttentionBlock(nn.Module):

    def __init__(self, embed_dim, hidden_dim, num_heads, dropout=0.0):
        """
        Inputs:
            embed_dim - Dimensionality of input and attention feature vectors
            hidden_dim - Dimensionality of hidden layer in feed-forward network
                        (usually 2-4x larger than embed_dim)
            num_heads - Number of heads to use in the Multi-Head Attention block
            dropout - Amount of dropout to apply in the feed-forward network
        """
        super().__init__()

        self.layer_norm_1 = nn.LayerNorm(embed_dim)
        self.attn = nn.MultiheadAttention(embed_dim, num_heads,
                                         dropout=dropout)
        self.layer_norm_2 = nn.LayerNorm(embed_dim)
        self.linear = nn.Sequential(
            nn.Linear(embed_dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, embed_dim),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        inp_x = self.layer_norm_1(x)
```

(continues on next page)

(continued from previous page)

```

x = x + self.attn(inp_x, inp_x, inp_x)[0]
x = x + self.linear(self.layer_norm_2(x))
return x

```

Now we have all modules ready to build our own Vision Transformer. Besides the Transformer encoder, we need the following modules:

- A **linear projection** layer that maps the input patches to a feature vector of larger size. It is implemented by a simple linear layer that takes each $M \times M$ patch independently as input.
- A **classification token** that is added to the input sequence. We will use the output feature vector of the classification token (CLS token in short) for determining the classification prediction.
- Learnable **positional encodings** that are added to the tokens before being processed by the Transformer. Those are needed to learn position-dependent information, and convert the set to a sequence. Since we usually work with a fixed resolution, we can learn the positional encodings instead of having the pattern of sine and cosine functions.
- An **MLP head** that takes the output feature vector of the CLS token, and maps it to a classification prediction. This is usually implemented by a small feed-forward network or even a single linear layer.

With those components in mind, let's implement the full Vision Transformer below:

```
[7]: class VisionTransformer(nn.Module):

    def __init__(self, embed_dim, hidden_dim, num_channels, num_heads, num_layers, num_
     ↪classes, patch_size, num_patches, dropout=0.0):
        """
        Inputs:
            embed_dim - Dimensionality of the input feature vectors to the Transformer
            hidden_dim - Dimensionality of the hidden layer in the feed-forward networks
                         within the Transformer
            num_channels - Number of channels of the input (3 for RGB)
            num_heads - Number of heads to use in the Multi-Head Attention block
            num_layers - Number of layers to use in the Transformer
            num_classes - Number of classes to predict
            patch_size - Number of pixels that the patches have per dimension
            num_patches - Maximum number of patches an image can have
            dropout - Amount of dropout to apply in the feed-forward network and
                     on the input encoding
        """
        super().__init__()

        self.patch_size = patch_size

        # Layers/Networks
        self.input_layer = nn.Linear(num_channels*(patch_size**2), embed_dim)
        self.transformer = nn.Sequential(*[AttentionBlock(embed_dim, hidden_dim, num_
     ↪heads, dropout=dropout) for _ in range(num_layers)])
        self.mlp_head = nn.Sequential(
            nn.LayerNorm(embed_dim),
            nn.Linear(embed_dim, num_classes)
        )
        self.dropout = nn.Dropout(dropout)
```

(continues on next page)

(continued from previous page)

```

# Parameters/Embeddings
self.cls_token = nn.Parameter(torch.randn(1,1,embed_dim))
self.pos_embedding = nn.Parameter(torch.randn(1,1+num_patches,embed_dim))

def forward(self, x):
    # Preprocess input
    x = img_to_patch(x, self.patch_size)
    B, T, _ = x.shape
    x = self.input_layer(x)

    # Add CLS token and positional encoding
    cls_token = self.cls_token.repeat(B, 1, 1)
    x = torch.cat([cls_token, x], dim=1)
    x = x + self.pos_embedding[:, :T+1]

    # Apply Transformer
    x = self.dropout(x)
    x = x.transpose(0, 1)
    x = self.transformer(x)

    # Perform classification prediction
    cls = x[0]
    out = self.mlp_head(cls)
    return out

```

Finally, we can put everything into a PyTorch Lightning Module as usual. We use `torch.optim.AdamW` as the optimizer, which is Adam with a corrected weight decay implementation. Since we use the Pre-LN Transformer version, we do not need to use a learning rate warmup stage anymore. Instead, we use the same learning rate scheduler as the CNNs in our previous tutorial on image classification.

```

[8]: class ViT(pl.LightningModule):

    def __init__(self, model_kwargs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.model = VisionTransformer(**model_kwargs)
        self.example_input_array = next(iter(train_loader))[0]

    def forward(self, x):
        return self.model(x)

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(), lr=self.hparams.lr)
        lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[100, 150], gamma=0.1)
        return [optimizer], [lr_scheduler]

    def _calculate_loss(self, batch, mode="train"):
        imgs, labels = batch
        preds = self.model(imgs)
        loss = F.cross_entropy(preds, labels)

```

(continues on next page)

(continued from previous page)

```

acc = (preds.argmax(dim=-1) == labels).float().mean()

self.log(f'{mode}_loss', loss)
self.log(f'{mode}_acc', acc)
return loss

def training_step(self, batch, batch_idx):
    loss = self._calculate_loss(batch, mode="train")
    return loss

def validation_step(self, batch, batch_idx):
    self._calculate_loss(batch, mode="val")

def test_step(self, batch, batch_idx):
    self._calculate_loss(batch, mode="test")

```

4.16.2 Experiments

Commonly, Vision Transformers are applied to large-scale image classification benchmarks such as ImageNet to leverage their full potential. However, here we take a step back and ask: can Vision Transformer also succeed on classical, small benchmarks such as CIFAR10? To find this out, we train a Vision Transformer from scratch on the CIFAR10 dataset. Let's first create a training function for our PyTorch Lightning module which also loads the pre-trained model if you have downloaded it above.

```

[9]: def train_model(**kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "ViT"),
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=180,
                          callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
                                                      monitor="val_acc"),
                                     LearningRateMonitor("epoch")])
    trainer.logger._log_graph = True          # If True, we plot the computation graph in
    # tensorboard
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't
    # need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "ViT.ckpt")
    if os.path.isfile(pretrained_filename):
        print(f"Found pretrained model at {pretrained_filename}, loading...")
        model = ViT.load_from_checkpoint(pretrained_filename) # Automatically loads the
    # model with the saved hyperparameters
    else:
        pl.seed_everything(42) # To be reproducible
        model = ViT(**kwargs)
        trainer.fit(model, train_loader, val_loader)
        model = ViT.load_from_checkpoint(trainer.checkpoint_callback.best_model_path) #_
    # Load best checkpoint after training

    # Test best model on validation and test set

```

(continues on next page)

(continued from previous page)

```

val_result = trainer.test(model, val_loader, verbose=False)
test_result = trainer.test(model, test_loader, verbose=False)
result = {"test": test_result[0]["test_acc"], "val": val_result[0]["test_acc"]}

return model, result

```

Now, we can already start training our model. As seen in our implementation, we have a couple of hyperparameters that we have to set. When creating this notebook, we have performed a small grid search over hyperparameters and listed the best hyperparameters in the cell below. Nevertheless, it is worth discussing the influence that each hyperparameter has, and what intuition we have for choosing its value.

First, let's consider the patch size. The smaller we make the patches, the longer the input sequences to the Transformer become. While in general, this allows the Transformer to model more complex functions, it requires a longer computation time due to its quadratic memory usage in the attention layer. Furthermore, small patches can make the task more difficult since the Transformer has to learn which patches are close-by, and which are far away. We experimented with patch sizes of 2, 4, and 8 which gives us the input sequence lengths of 256, 64, and 16 respectively. We found 4 to result in the best performance and hence pick it below.

Next, the embedding and hidden dimensionality have a similar impact on a Transformer as to an MLP. The larger the sizes, the more complex the model becomes, and the longer it takes to train. In Transformers, however, we have one more aspect to consider: the query-key sizes in the Multi-Head Attention layers. Each key has the feature dimensionality of `embed_dim/num_heads`. Considering that we have an input sequence length of 64, a minimum reasonable size for the key vectors is 16 or 32. Lower dimensionalities can restrain the possible attention maps too much. We observed that more than 8 heads are not necessary for the Transformer, and therefore pick an embedding dimensionality of 256. The hidden dimensionality in the feed-forward networks is usually 2-4x larger than the embedding dimensionality, and thus we pick 512.

Finally, the learning rate for Transformers is usually relatively small, and in papers, a common value to use is 3e-5. However, since we work with a smaller dataset and have a potentially easier task, we found that we are able to increase the learning rate to 3e-4 without any problems. To reduce overfitting, we use a dropout value of 0.2. Remember that we also use small image augmentations as regularization during training.

Feel free to explore the hyperparameters yourself by changing the values below. In general, the Vision Transformer did not show to be too sensitive to the hyperparameter choices on the CIFAR10 dataset.

```
[10]: model, results = train_model(model_kwargs={
    'embed_dim': 256,
    'hidden_dim': 512,
    'num_heads': 8,
    'num_layers': 6,
    'patch_size': 4,
    'num_channels': 3,
    'num_patches': 64,
    'num_classes': 10,
    'dropout': 0.2
},
lr=3e-4)
print("ViT results", results)

GPU available: True, used: True
TPU available: False, using: 0 TPU cores

Found pretrained model at ../saved_models/tutorial15/ViT.ckpt, loading...

LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
```

```
HBox(children=(HTML(value='Testing'), FloatProgress(value=1.0, bar_style='info',  
    ↪layout=Layout(flex='2'), max=...
```

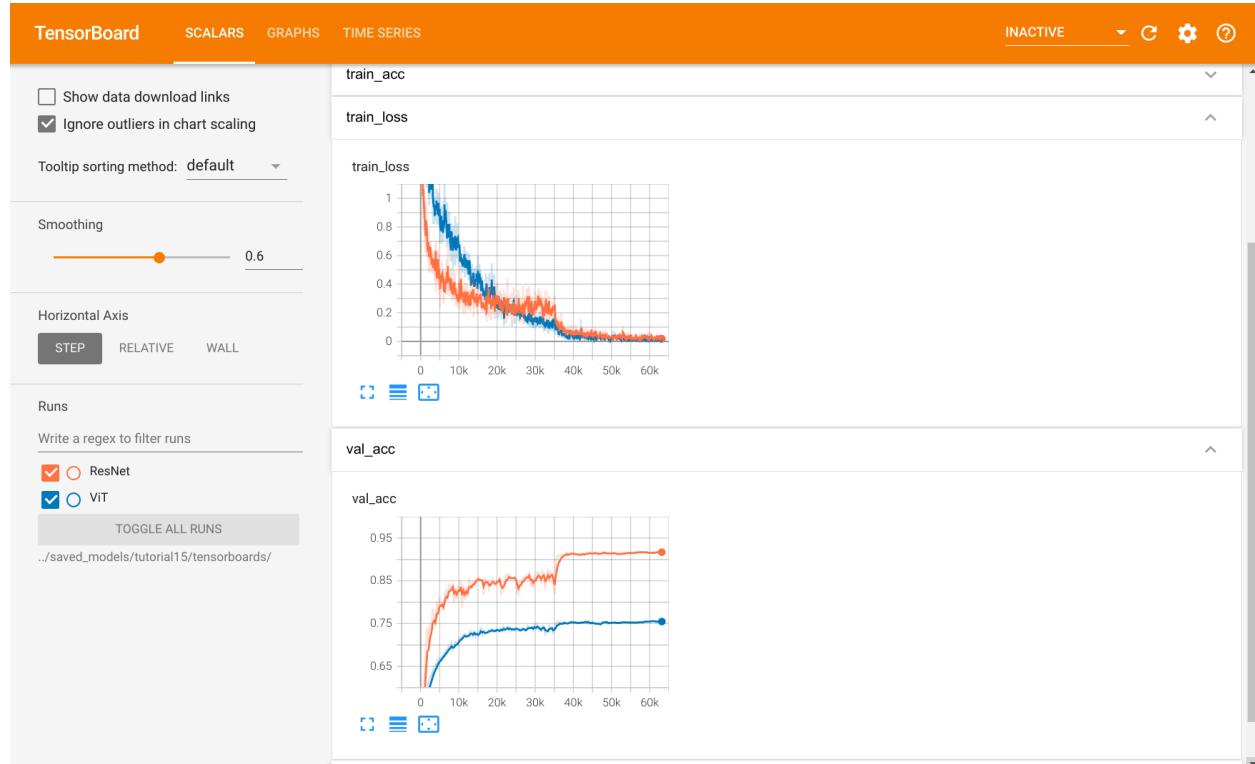
```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
```

```
HBox(children=(HTML(value='Testing'), FloatProgress(value=1.0, bar_style='info',  
    ↪layout=Layout(flex='2'), max=...
```

```
ViT results {'test': 0.7559000253677368, 'val': 0.7563999891281128}
```

The Vision Transformer achieves a validation and test performance of about 75%. In comparison, almost all CNN architectures that we have tested in [Tutorial 5](#) obtained a classification performance of around 90%. This is a considerable gap and shows that although Vision Transformers perform strongly on ImageNet with potential pretraining, they cannot come close to simple CNNs on CIFAR10 when being trained from scratch. The differences between a CNN and Transformer can be well observed in the training curves. Let's look at them in a tensorboard below:

[11]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH!
%tensorboard --logdir ../saved_models/tutorial15/tensorboards/



The tensorboard compares the Vision Transformer to a ResNet trained on CIFAR10. When looking at the training losses, we see that the ResNet learns much more quickly in the first iterations. While the learning rate might have an influence on the initial learning speed, we see the same trend in the validation accuracy. The ResNet achieves the best performance of the Vision Transformer after just 5 epochs (2000 iterations). Further, while the ResNet training loss and validation accuracy have a similar trend, the validation performance of the Vision Transformers only marginally changes after 10k iterations while the training loss has almost just started going down. Yet, the Vision Transformer is also able to achieve close to 100% accuracy on the training set.

All those observed phenomenons can be explained with a concept that we have visited before: inductive biases. Convolutional Neural Networks have been designed with the assumption that images are translation invariant. Hence, we

apply convolutions with shared filters across the image. Furthermore, a CNN architecture integrates the concept of distance in an image: two pixels that are close to each other are more related than two distant pixels. Local patterns are combined into larger patterns until we perform our classification prediction. All those aspects are inductive biases of a CNN. In contrast, a Vision Transformer does not know which two pixels are close to each other, and which are far apart. It has to learn this information solely from the sparse learning signal of the classification task. This is a huge disadvantage when we have a small dataset since such information is crucial for generalizing to an unseen test dataset. With large enough datasets and/or good pre-training, a Transformer can learn this information without the need for inductive biases, and instead is more flexible than a CNN. Especially long-distance relations between local patterns can be difficult to process in CNNs, while in Transformers, all patches have the distance of one. This is why Vision Transformers are so strong on large-scale datasets such as ImageNet but underperform a lot when being applied to a small dataset such as CIFAR10.

4.16.3 Conclusion

In this tutorial, we have implemented our own Vision Transformer from scratch and applied it to the task of image classification. Vision Transformers work by splitting an image into a sequence of smaller patches, use those as input to a standard Transformer encoder. While Vision Transformers achieved outstanding results on large-scale image recognition benchmarks such as ImageNet, they considerably underperform when being trained from scratch on small-scale datasets like CIFAR10. The reason is that in contrast to CNNs, Transformers do not have the inductive biases of translation invariance and the feature hierarchy (i.e. larger patterns consist of many smaller patterns). However, these aspects can be learned when enough data is provided, or the model has been pre-trained on other large-scale tasks. Considering that Vision Transformers have just been proposed end of 2020, there is likely a lot more to come on Transformers for Computer Vision.

References

- Dosovitskiy, Alexey, et al. “An image is worth 16x16 words: Transformers for image recognition at scale.” International Conference on Learning Representations (2021). [link](#)
- Chen, Xiangning, et al. “When Vision Transformers Outperform ResNets without Pretraining or Strong Data Augmentations.” arXiv preprint arXiv:2106.01548 (2021). [link](#)
- Tolstikhin, Ilya, et al. “MLP-mixer: An all-MLP Architecture for Vision.” arXiv preprint arXiv:2105.01601 (2021). [link](#)
- Xiong, Ruibin, et al. “On layer normalization in the transformer architecture.” International Conference on Machine Learning. PMLR, 2020. [link](#)
-

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.17 Tutorial 16: Meta-Learning - Learning to Learn

Filled notebook:

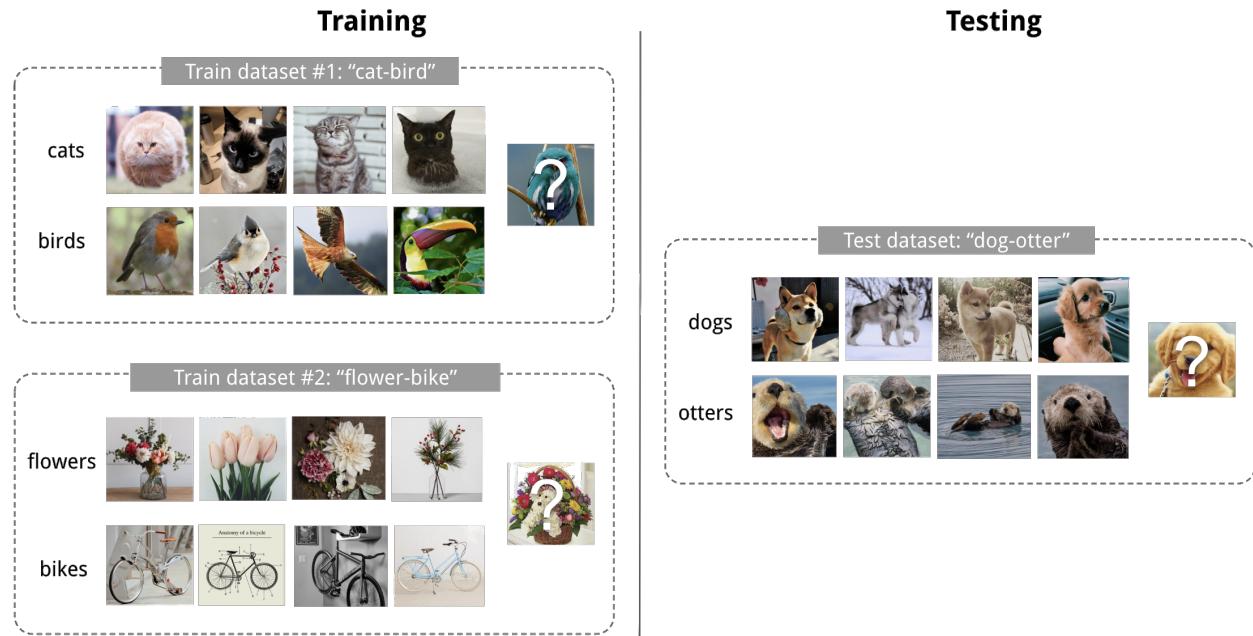
Pre-trained models:

Recordings:

Author: Phillip Lippe

In this tutorial, we will discuss algorithms that learn models which can quickly adapt to new classes and/or tasks with few samples. This area of machine learning is called *Meta-Learning* aiming at “learning to learn”. Learning from very few examples is a natural task for humans. In contrast to current deep learning models, we need to see only a few examples of a police car or firetruck to recognize them in daily traffic. This is a crucial ability since, in real-world applications, it is rarely the case that the data stays static and does not change over time. For example, an object detection system for mobile phones trained on data from 2000 will have trouble detecting today’s common mobile phones, and thus, needs to adapt to new data without excessive label effort. The optimization techniques we have discussed so far struggle with this because they only aim at obtaining a good performance on a test set that had similar data. However, what if the test set has classes that we do not have in the training set? Or what if we want to test the model on a completely different task?

Meta-Learning offers solutions to these situations, and we will discuss three popular algorithms: **Prototypical Networks** (Snell et al., 2017), **Model-Agnostic Meta-Learning / MAML** (Finn et al., 2017), and **Proto-MAML** (Triantafillou et al., 2020). We will focus on the task of few-shot classification where the training and test set have distinct sets of classes. For instance, we would train the model on the binary classifications of cats-birds and flowers-bikes, but during test time, the model would need to learn from 4 examples each the difference between dogs and otters, two classes we have not seen during training (Figure credit - [Lilian Weng](#)).



A different setup, which is very common in Reinforcement Learning and recently Natural Language Processing, is to aim at few-shot learning of a completely new task. For example, a robot agent that learned to run, jump and pick up boxes, should quickly adapt to collecting and stacking boxes. In NLP, we can think of a model which was trained

in sentiment classification, hate speech detection, and sarcasm classification, to adapt to classifying the emotion of a text. All methods we will discuss in this notebook can be easily applied to these settings since we only use a different definition of a ‘task’. For few-shot classification, we consider a task to distinguish between M novel classes. Here, we would not only have novel classes, but also a completely different dataset.

First of all, let’s start with importing our standard libraries. We will again be using PyTorch Lightning.

```
[1]: ## Standard libraries
import os
import numpy as np
import random
import json
from PIL import Image
from collections import defaultdict
from statistics import mean, stdev
from copy import deepcopy

## Imports for plotting
import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## tqdm for loading bars
from tqdm.auto import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim

## Torchvision
import torchvision
from torchvision.datasets import CIFAR100, SVHN
from torchvision import transforms

# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Import tensorboard
%load_ext tensorboard
```

(continues on next page)

(continued from previous page)

```
# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial16"

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", device)

Global seed set to 42
Device: cuda:0
```

Training the models in this notebook can take between 2 and 8 hours, and the evaluation time of some algorithms is in the span of couples of minutes. Hence, we download pre-trained models and results below.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial16/"
# Files to download
pretrained_files = ["ProtoNet.ckpt", "ProtoMAML.ckpt",
                    "tensorboards/ProtoNet/events.out.tfevents.ProtoNet",
                    "tensorboards/ProtoMAML/events.out.tfevents.ProtoMAML",
                    "protomaml_fewshot.json",
                    "protomaml_svhn_fewshot.json"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive ↵
            ↵folder, or contact the author with the full output including the following error:\n", ↵
            ↵e)
```

4.17.1 Few-shot classification

We start our implementation by discussing the dataset setup. In this notebook, we will use CIFAR100 which we have already seen in Tutorial 6. CIFAR100 has 100 classes each with 600 images of size 32×32 pixels. Instead of splitting the training, validation, and test set over examples, we will split them over classes: we will use 80 classes for training, and 10 for validation, and 10 for testing. Our overall goal is to obtain a model that can distinguish between the 10 test classes with seeing very few examples. First, let's load the dataset and visualize some examples.

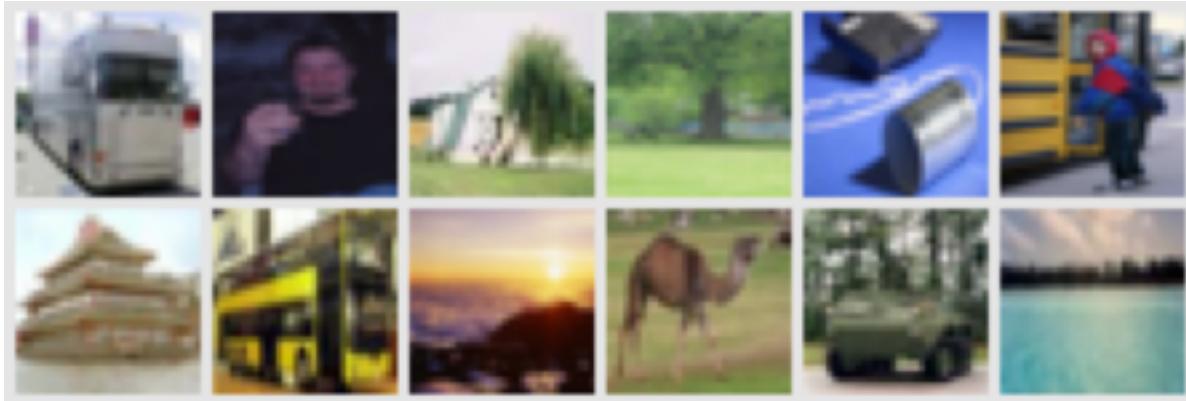
```
[3]: # Loading CIFAR100 dataset
CIFAR_train_set = CIFAR100(root=DATASET_PATH, train=True, download=True,
                           transform=transforms.ToTensor())
CIFAR_test_set = CIFAR100(root=DATASET_PATH, train=False, download=True,
                           transform=transforms.ToTensor())
```

Files already downloaded and verified
 Files already downloaded and verified

```
[4]: # Visualize some examples
NUM_IMAGES = 12
CIFAR_images = torch.stack([CIFAR_train_set[np.random.randint(len(CIFAR_train_set))][0]
                           for idx in range(NUM_IMAGES)], dim=0)
img_grid = torchvision.utils.make_grid(CIFAR_images, nrow=6, normalize=True, pad_value=0.
                                       _9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("Image examples of the CIFAR100 dataset")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

Image examples of the CIFAR100 dataset



Data preprocessing

Next, we need to prepare the dataset in the training, validation and test split as mentioned before. The torchvision package gives us the training and test set as two separate dataset objects. The next code cells will merge the original training and test set, and then create the new train-val-test split.

```
[5]: # Merging original training and test set
CIFAR_all_images = np.concatenate([CIFAR_train_set.data, CIFAR_test_set.data], axis=0)
CIFAR_all_targets = torch.LongTensor(CIFAR_train_set.targets + CIFAR_test_set.targets)
```

To have an easier time handling the dataset, we define our own, simple dataset class below. It takes a set of images, labels/targets, and image transformations, and returns the corresponding images and labels element-wise.

```
[6]: class ImageDataset(data.Dataset):

    def __init__(self, imgs, targets, img_transform=None):
        """
        Inputs:
            imgs - Numpy array of shape [N, 32, 32, 3] containing all images.
            targets - PyTorch array of shape [N] containing all labels.
            img_transform - A torchvision transformation that should be applied
                           to the images before returning. If none, no transformation
                           is applied.
        """
        super().__init__()
        self.img_transform = img_transform
        self.imgs = imgs
        self.targets = targets

    def __getitem__(self, idx):
        img, target = self.imgs[idx], self.targets[idx]
        img = Image.fromarray(img)

        if self.img_transform is not None:
            img = self.img_transform(img)

        return img, target

    def __len__(self):
        return self.imgs.shape[0]
```

Now, we can create the class splits. We will assign the classes randomly to training, validation and test, and use a 80%-10%-10% split.

```
[7]: torch.manual_seed(0)          # Set seed for reproducibility
classes = torch.randperm(100)    # Returns random permutation of numbers 0 to 99
train_classes, val_classes, test_classes = classes[:80], classes[80:90], classes[90:]
```

To get an intuition of the validation and test classes, we print the class names below:

```
[8]: # Printing validation and test classes
idx_to_class = {val: key for key, val in CIFAR_train_set.class_to_idx.items()}
print("Validation classes:", [idx_to_class[c.item()] for c in val_classes])
print("Test classes:", [idx_to_class[c.item()] for c in test_classes])
```

```
Validation classes: ['caterpillar', 'castle', 'skunk', 'ray', 'bus', 'motorcycle',
↳ 'keyboard', 'chimpanzee', 'possum', 'tiger']
Test classes: ['kangaroo', 'crocodile', 'butterfly', 'shark', 'forest', 'pickup_truck',
↳ 'telephone', 'lion', 'worm', 'mushroom']
```

As we can see, the classes have quite some variety and some classes might be easier to distinguish than others. For instance, in the test classes, ‘pickup_truck’ is the only vehicle while the classes ‘mushroom’, ‘worm’ and ‘forest’ might be harder to keep apart. Remember that we want to learn the classification of those ten classes from 80 other classes in our training set, and few examples from the actual test classes. We will experiment with the number of examples per class.

Finally, we can create the training, validation and test dataset according to our split above. For this, we create dataset objects of our previously defined class `ImageDataset`.

```
[9]: def dataset_from_labels(imgs, targets, class_set, **kwargs):
    class_mask = (targets[:,None] == class_set[None,:]).any(dim=-1)
    return ImageDataset(imgs=imgs[class_mask],
                        targets=targets[class_mask],
                        **kwargs)
```

As in our experiments before on CIFAR in Tutorial 5, 6 and 9, we normalize the dataset. Additionally, we use small augmentations during training to prevent overfitting.

```
[10]: # Pre-computed statistics from the new train set
DATA_MEANS = torch.Tensor([0.5183975 , 0.49192241, 0.44651328])
DATA_STD = torch.Tensor([0.26770132, 0.25828985, 0.27961241])

test_transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize(
                                        DATA_MEANS, DATA_STD)
                                   ])
# For training, we add some augmentation.
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                      transforms.RandomResizedCrop(
                                          (32, 32), scale=(0.8, 1.0), ratio=(0.9, 1.1)),
                                      transforms.ToTensor(),
                                      transforms.Normalize(
                                          DATA_MEANS, DATA_STD)
                                     ])

train_set = dataset_from_labels(
    CIFAR_all_images, CIFAR_all_targets, train_classes, img_transform=train_transform)
val_set = dataset_from_labels(
    CIFAR_all_images, CIFAR_all_targets, val_classes, img_transform=test_transform)
test_set = dataset_from_labels(
    CIFAR_all_images, CIFAR_all_targets, test_classes, img_transform=test_transform)
```

Data sampling

The strategy of how to use the available training data for learning few-shot adaptation is crucial in meta-learning. All three algorithms that we discuss here have a similar idea: simulate few-shot learning during training. Specifically, at each training step, we randomly select a small number of classes and sample a small number of examples for each class. This represents our few-shot training batch, which we also refer to as **support set**. Additionally, we sample a second set of examples from the same classes and refer to this batch as **query set**. Our training objective is to classify the query set correctly from seeing the support set and its corresponding labels. The main difference between our three methods (ProtoNet, MAML, and Proto-MAML) is in how they use the support set to adapt to the training classes.

This subsection summarizes the code that is needed to create such training batches. In PyTorch, we can specify the data sampling procedure by so-called `Sampler` ([documentation](#)). Samplers are iterable objects that return indices in the order in which the data elements should be sampled. In our previous notebooks, we usually used the option `shuffle=True` in the `data.DataLoader` objects which creates a sampler returning the data indices in random order. Here, we focus on samplers that return batches of indices that correspond to support and query set batches. Below, we implement such a sampler.

```
[11]: class FewShotBatchSampler(object):

    def __init__(self, dataset_targets, N_way, K_shot, include_query=False, shuffle=True,
    ↵ shuffle_once=False):
        """
        Inputs:
            dataset_targets - PyTorch tensor of the labels of the data elements.
            N_way - Number of classes to sample per batch.
            K_shot - Number of examples to sample per class in the batch.
            include_query - If True, returns batch of size N_way*K_shot*2, which
                            can be split into support and query set. Simplifies
                            the implementation of sampling the same classes but
                            distinct examples for support and query set.
            shuffle - If True, examples and classes are newly shuffled in each
                      iteration (for training)
            shuffle_once - If True, examples and classes are shuffled once in
                           the beginning, but kept constant across iterations
                           (for validation)
        """
        super().__init__()
        self.dataset_targets = dataset_targets
        self.N_way = N_way
        self.K_shot = K_shot
        self.shuffle = shuffle
        self.include_query = include_query
        if self.include_query:
            self.K_shot *= 2
        self.batch_size = self.N_way * self.K_shot # Number of overall images per batch

        # Organize examples by class
        self.classes = torch.unique(self.dataset_targets).tolist()
        self.num_classes = len(self.classes)
        self.indices_per_class = {}
        self.batches_per_class = {} # Number of K-shot batches that each class can
    ↵ provide
        for c in self.classes:
            self.indices_per_class[c] = torch.where(self.dataset_targets == c)[0]
```

(continues on next page)

(continued from previous page)

```

    self.batches_per_class[c] = self.indices_per_class[c].shape[0] // self.K_shot

    # Create a list of classes from which we select the N classes per batch
    self.iterations = sum(self.batches_per_class.values()) // self.N_way
    self.class_list = [c for c in self.classes for _ in range(self.batches_per_
→class[c])]

    if shuffle_once or self.shuffle:
        self.shuffle_data()
    else:
        # For testing, we iterate over classes instead of shuffling them
        sort_idxs = [i+p*self.num_classes for i,
                     c in enumerate(self.classes) for p in range(self.batches_per_
→class[c])]
        self.class_list = np.array(self.class_list)[np.argsort(sort_idxs)].tolist()

    def shuffle_data(self):
        # Shuffle the examples per class
        for c in self.classes:
            perm = torch.randperm(self.indices_per_class[c].shape[0])
            self.indices_per_class[c] = self.indices_per_class[c][perm]
        # Shuffle the class list from which we sample. Note that this way of shuffling
        # does not prevent to choose the same class twice in a batch. However, for
        # training and validation, this is not a problem.
        random.shuffle(self.class_list)

    def __iter__(self):
        # Shuffle data
        if self.shuffle:
            self.shuffle_data()

        # Sample few-shot batches
        start_index = defaultdict(int)
        for it in range(self.iterations):
            class_batch = self.class_list[it*self.N_way:(it+1)*self.N_way] # Select N_
→classes for the batch
            index_batch = []
            for c in class_batch: # For each class, select the next K examples and add_
→them to the batch
                index_batch.extend(self.indices_per_class[c][start_index[c]:start_
→index[c]+self.K_shot])
                start_index[c] += self.K_shot
            if self.include_query: # If we return support+query set, sort them so that_
→they are easy to split
                index_batch = index_batch[::-2] + index_batch[1::2]
            yield index_batch

    def __len__(self):
        return self.iterations

```

Note that this sampler eventually allows the sampling of batches where we use a class twice due to a simpler shuffling function (`shuffle_data`). In other words, during training or validation, if we sample batches for a 5-class 4-shot training setting, it can occasionally happen that we get a batch where 2 of the 5 classes are identical. Since we have 80 classes to choose from, this however happens relatively rarely. Further, it actually does not constitute any issue if

the code for the meta-learning methods support variable number of classes and shots per class. Nonetheless, when the number of classes/tasks is smaller, it is recommended to replace the `shuffle_data` method with a sampler that prevents picking the same class twice in a batch.

Now, we can create our intended data loaders by passing an object of `FewShotBatchSampler` as `batch_sampler=...` input to the PyTorch data loader object. For our experiments, we will use a 5-class 4-shot training setting. This means that each support set contains 5 classes with 4 examples each, i.e., 20 images overall. Usually, it is good to keep the number of shots equal to the number that you aim to test on. However, we will experiment later with a different number of shots, and hence, we pick 4 as a compromise for now. To get the best-performing model, it is recommended to consider the number of training shots as hyperparameters in a grid search.

```
[12]: N_WAY = 5
K_SHOT = 4
train_data_loader = data.DataLoader(train_set,
                                    batch_sampler=FewShotBatchSampler(train_set.targets,
                                                                    include_query=True,
                                                                    N_way=N_WAY,
                                                                    K_shot=K_SHOT,
                                                                    shuffle=True),
                                    num_workers=4)
val_data_loader = data.DataLoader(val_set,
                                    batch_sampler=FewShotBatchSampler(val_set.targets,
                                                                    include_query=True,
                                                                    N_way=N_WAY,
                                                                    K_shot=K_SHOT,
                                                                    shuffle=False,
                                                                    shuffle_once=True),
                                    num_workers=4)
```

For simplicity, we implemented the sampling of a support and query set as sampling a support set with twice the number of examples. After sampling a batch from the data loader, we need to split it into a support and query set. We can summarize this step in the following function:

```
[13]: def split_batch(imgs, targets):
    support_imgs, query_imgs = imgs.chunk(2, dim=0)
    support_targets, query_targets = targets.chunk(2, dim=0)
    return support_imgs, query_imgs, support_targets, query_targets
```

Finally, to ensure that our implementation of the data sampling process is correct, we can sample a batch and visualize its support and query set. What we would like to see is that the support and query set have the same classes, but distinct examples.

```
[14]: imgs, targets = next(iter(val_data_loader)) # We use the validation set since it does not apply augmentations
support_imgs, query_imgs, _, _ = split_batch(imgs, targets)
support_grid = torchvision.utils.make_grid(support_imgs, nrow=K_SHOT, normalize=True, pad_value=0.9)
support_grid = support_grid.permute(1, 2, 0)
query_grid = torchvision.utils.make_grid(query_imgs, nrow=K_SHOT, normalize=True, pad_value=0.9)
query_grid = query_grid.permute(1, 2, 0)

fig, ax = plt.subplots(1, 2, figsize=(8, 5))
ax[0].imshow(support_grid)
```

(continues on next page)

(continued from previous page)

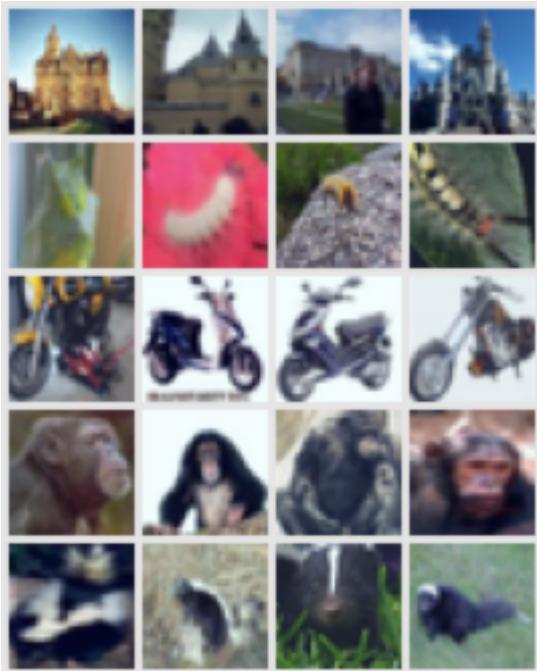
```

ax[0].set_title("Support set")
ax[0].axis('off')
ax[1].imshow(query_grid)
ax[1].set_title("Query set")
ax[1].axis('off')
plt.suptitle("Few Shot Batch", weight='bold')
plt.show()
plt.close()

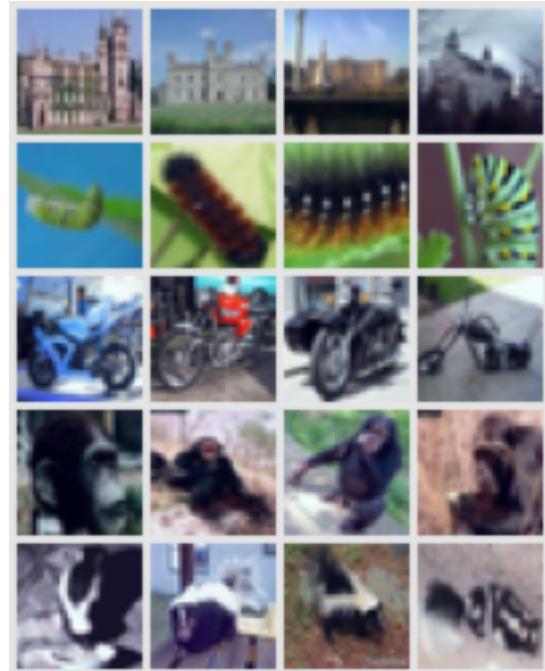
```

Few Shot Batch

Support set



Query set



As we can see, the support and query set have the same five classes, but different examples. The models will be tasked to classify the examples in the query set by learning from the support set and its labels. With the data sampling in place, we can now start to implement our first meta-learning model: Prototypical Networks.

4.17.2 Prototypical Networks

The Prototypical Network, or ProtoNet for short, is a metric-based meta-learning algorithm that operates similarly to the nearest neighbor classification. Metric-based meta-learning methods classify a new example \mathbf{x} based on some distance function d_φ between x and all elements in the support set. ProtoNets implements this idea with the concept of prototypes in a learned feature space. First, ProtoNet uses an embedding function f_θ to encode each input in the support set into a L -dimensional feature vector. Next, for each class c , we collect the feature vectors of all examples with label c and average their feature vectors. Formally, we can define this as:

$$\mathbf{v}_c = \frac{1}{|S_c|} \sum_{(\mathbf{x}_i, y_i) \in S_c} f_\theta(\mathbf{x}_i)$$

where S_c is the part of the support set S for which $y_i = c$, and \mathbf{v}_c represents the *prototype* of class c . The prototype calculation is visualized below for a 2-dimensional feature space and 3 classes (Figure credit - Snell et al.). The colored dots represent encoded support elements with the color-corresponding class labels, and the black dots next to the class label are the averaged prototypes.

Based on these prototypes, we want to classify a new example. Remember that since we want to learn the encoding function f_θ , this classification must be differentiable, and hence, we need to define a probability distribution across classes. For this, we will make use of the distance function d_φ : the closer a new example \mathbf{x} is to a prototype \mathbf{v}_c , the higher the probability for \mathbf{x} belonging to class c . Formally, we can simply use a softmax over the distances of \mathbf{x} to all class prototypes:

$$p(y = c|\mathbf{x}) = \text{softmax}(-d_\varphi(f_\theta(\mathbf{x}), \mathbf{v}_c)) = \frac{\exp(-d_\varphi(f_\theta(\mathbf{x}), \mathbf{v}_c))}{\sum_{c' \in \mathcal{C}} \exp(-d_\varphi(f_\theta(\mathbf{x}), \mathbf{v}_{c'}))}$$

Note that the negative sign is necessary since we want to increase the probability for close-by vectors and have a low probability for distant vectors. We train the network f_θ based on the cross-entropy error of the training query set examples. Thereby, the gradient flows through both the prototypes \mathbf{v}_c and the query set encodings $f_\theta(\mathbf{x})$. For the distance function d_φ , we can choose any function as long as it is differentiable concerning both of its inputs. The most common function, which we also use here, is the squared euclidean distance, but there have been several works on different distance functions as well.

ProtoNet implementation

Now that we know how a ProtoNet works in principle, let's look at how we can apply it to our specific problem of few-shot image classification, and implement it below. First, we need to define the encoder function f_θ . Since we work with CIFAR images, we can take a look back at Tutorial 5 where we compared common Computer Vision architectures, and choose one of the best-performing ones. Here, we go with a DenseNet since it is in general more parameter efficient than ResNet. Luckily, we do not need to implement DenseNet ourselves again and can rely on torchvision's model package instead. We use common hyperparameters of 64 initial feature channels, add 32 per block, and use a bottleneck size of 64 (i.e. 2 times the growth rate). We use 4 stages of 6 layers each, which results in overall about 1 million parameters. Note that the torchvision package assumes that the last layer is used for classification and hence calls its output size `num_classes`. However, we can instead just use it as the feature space of ProtoNet and choose an arbitrary dimensionality. We will use the same network for other algorithms in this notebook to ensure a fair comparison.

```
[15]: def get_convnet(output_size):
    convnet = torchvision.models.DenseNet(growth_rate=32,
                                          block_config=(6, 6, 6, 6),
                                          bn_size=2,
                                          num_init_features=64,
                                          num_classes=output_size # Output
                                          #dimensionality
                                          )
    return convnet
```

Next, we can look at implementing ProtoNet. We will define it as PyTorch Lightning module to use all functionalities of PyTorch Lightning. The first step during training is to encode all images in a batch with our network. Next, we calculate the class prototypes from the support set (function `calculate_prototypes`), and classify the query set examples according to the prototypes (function `classify_feats`). Keep in mind that we use the data sampling described before, such that the support and query set are stacked together in the batch. Thus, we use our previously defined function `split_batch` to split them apart. The full code can be found below.

```
[16]: class ProtoNet(pl.LightningModule):
```

(continues on next page)

(continued from previous page)

```

def __init__(self, proto_dim, lr):
    """
    Inputs
        proto_dim - Dimensionality of prototype feature space
        lr - Learning rate of Adam optimizer
    """
    super().__init__()
    self.save_hyperparameters()
    self.model = get_convnet(output_size=self.hparams.proto_dim)

def configure_optimizers(self):
    optimizer = optim.AdamW(self.parameters(), lr=self.hparams.lr)
    scheduler = optim.lr_scheduler.MultiStepLR(
        optimizer, milestones=[140, 180], gamma=0.1)
    return [optimizer], [scheduler]

@staticmethod
def calculate_prototypes(features, targets):
    # Given a stack of features vectors and labels, return class prototypes
    # features - shape [N, proto_dim], targets - shape [N]
    classes, _ = torch.unique(targets).sort() # Determine which classes we have
    prototypes = []
    for c in classes:
        p = features[torch.where(targets == c)[0]].mean(dim=0) # Average class_
    ↵feature vectors
        prototypes.append(p)
    prototypes = torch.stack(prototypes, dim=0)
    # Return the 'classes' tensor to know which prototype belongs to which class
    return prototypes, classes

def classify_feats(self, prototypes, classes, feats, targets):
    # Classify new examples with prototypes and return classification error
    dist = torch.pow(prototypes[None, :], 2) - feats[:, None, 2].sum(dim=2) # Squared_
    ↵euclidean distance
    preds = F.log_softmax(-dist, dim=1)
    labels = (classes[None, :] == targets[:, None]).long().argmax(dim=-1)
    acc = (preds.argmax(dim=1) == labels).float().mean()
    return preds, labels, acc

def calculate_loss(self, batch, mode):
    # Determine training loss for a given support and query set
    imgs, targets = batch
    features = self.model(imgs) # Encode all images of support and query set
    support_feats, query_feats, support_targets, query_targets = split_
    ↵batch(features, targets)
    prototypes, classes = ProtoNet.calculate_prototypes(support_feats, support_
    ↵targets)
    preds, labels, acc = self.classify_feats(prototypes, classes, query_feats, query_
    ↵targets)
    loss = F.cross_entropy(preds, labels)

    self.log(f"{mode}_loss", loss)

```

(continues on next page)

(continued from previous page)

```

    self.log(f"{{mode}}_acc", acc)
    return loss

def training_step(self, batch, batch_idx):
    return self.calculate_loss(batch, mode="train")

def validation_step(self, batch, batch_idx):
    _ = self.calculate_loss(batch, mode="val")

```

For validation, we use the same principle as training and sample support and query sets from the hold-out 10 classes. However, this gives us noisy scores depending on which query sets are chosen to which support sets. This is why we will use a different strategy during testing. For validation, our training strategy is sufficient since it is much faster than testing, and gives a good estimate of the training generalization as long as we keep the support-query sets constant across validation iterations.

Training

After implementing the model, we can already start training it. We use our common PyTorch Lightning training function, and train the model for 200 epochs. The training function takes `model_class` as input argument, i.e. the PyTorch Lightning module class that should be trained, since we will reuse this function for other algorithms as well.

```
[17]: def train_model(model_class, train_loader, val_loader, **kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, model_class._name_),
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=200,
                          callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
                                                      monitor="val_acc"),
                                     LearningRateMonitor("epoch")],
                          enable_progress_bar=False)
    trainer.logger._default_hp_metric = None

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(
        CHECKPOINT_PATH, model_class._name_ + ".ckpt")
    if os.path.isfile(pretrained_filename):
        print(f"Found pretrained model at {pretrained_filename}, loading...")
        # Automatically loads the model with the saved hyperparameters
        model = model_class.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything(42) # To be reproducible
        model = model_class(**kwargs)
        trainer.fit(model, train_loader, val_loader)
        model = model_class.load_from_checkpoint(
            trainer.checkpoint_callback.best_model_path) # Load best checkpoint after
                                                       # training

    return model
```

Below is the training call for our ProtoNet. We use a 64-dimensional feature space. Larger feature spaces showed to give noisier results since the squared euclidean distance becomes proportionally larger in expectation, and smaller

feature spaces might not allow for enough flexibility. We recommend to load the pre-trained model here at first, but feel free to play around with the hyperparameters yourself.

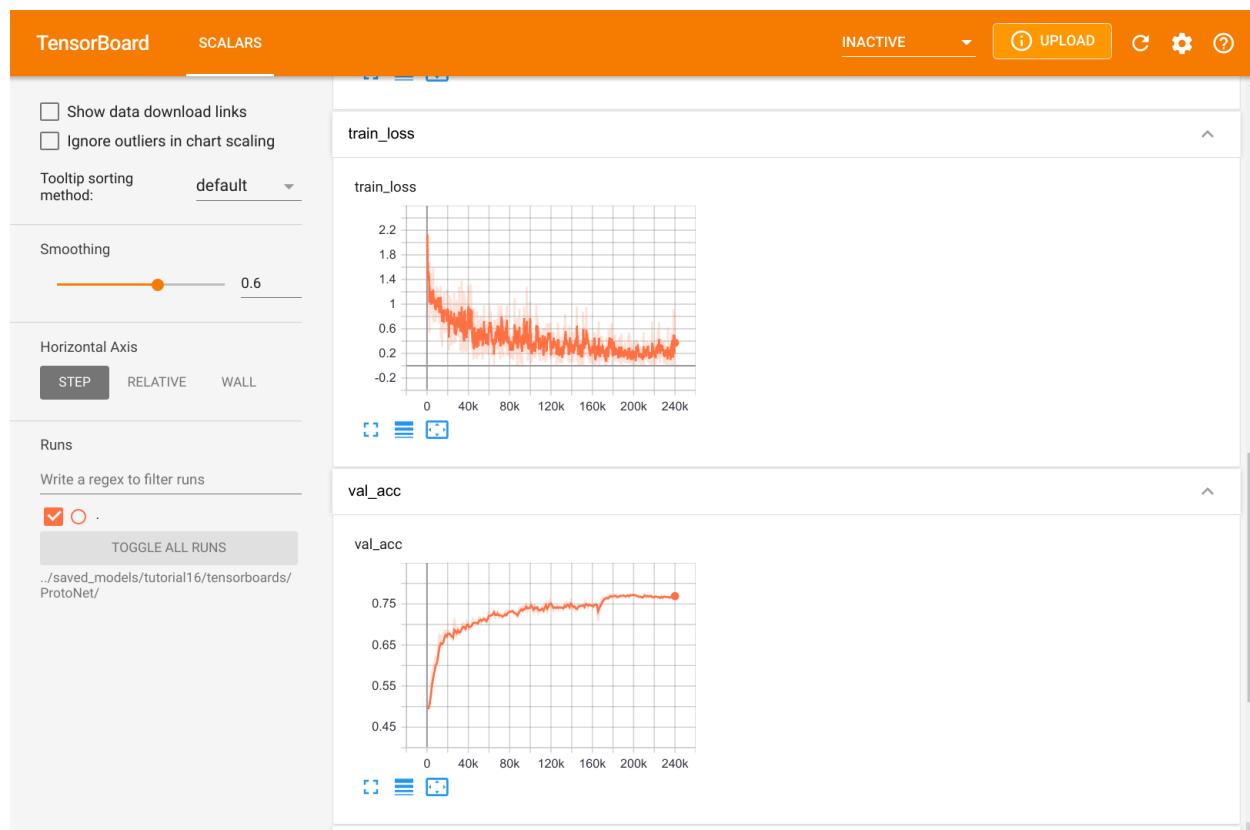
```
[18]: protonet_model = train_model(ProtoNet,
                                 proto_dim=64,
                                 lr=2e-4,
                                 train_loader=train_data_loader,
                                 val_loader=val_data_loader)
```

GPU available: True, used: True
 TPU available: False, using: 0 TPU cores

Found pretrained model at ../saved_models/tutorial16/ProtoNet.ckpt, loading...

We can also take a closer look at the TensorBoard below.

```
[19]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH if needed
%tensorboard --logdir ../saved_models/tutorial16/tensorboards/ProtoNet/
```



In contrast to standard supervised learning, we see that ProtoNet does not overfit as much as we would expect. The validation accuracy is of course lower than the average training, but the training loss does not stick close to zero. This is because no training batch is as the other, and we also mix new examples in the support set and query set. This gives us slightly different prototypes in every iteration and makes it harder for the network to fully overfit.

Testing

Our goal of meta-learning is to obtain a model that can quickly adapt to a new task, or in this case, new classes to distinguish between. To test this, we will use our trained ProtoNet and adapt it to the 10 test classes. Thereby, we pick k examples per class from which we determine the prototypes and test the classification accuracy on all other examples. This can be seen as using the k examples per class as a support set, and the rest of the dataset as a query set. We iterate through the dataset such that each example has been once included in a support set. The average performance across all support sets tells us how well we can expect ProtoNet to perform when seeing only k examples per class. During training, we used $k = 4$. In testing, we will experiment with $k = \{2, 4, 8, 16, 32\}$ to get a better sense of how k influences the results. We would expect that we achieve higher accuracies the more examples we have in the support set, but we don't know how it scales. Hence, let's first implement a function that executes the testing procedure for a given k :

```
[20]: @torch.no_grad()
def test_proto_net(model, dataset, data_feats=None, k_shot=4):
    """
    Inputs
        model - Pretrained ProtoNet model
        dataset - The dataset on which the test should be performed.
                  Should be instance of ImageDataset
        data_feats - The encoded features of all images in the dataset.
                    If None, they will be newly calculated, and returned
                    for later usage.
        k_shot - Number of examples per class in the support set.
    """
    model = model.to(device)
    model.eval()
    num_classes = dataset.targets.unique().shape[0]
    exmps_per_class = dataset.targets.shape[0]//num_classes # We assume uniform example_
    ↪distribution here

    # The encoder network remains unchanged across k-shot settings. Hence, we only need
    # to extract the features for all images once.
    if data_feats is None:
        # Dataset preparation
        dataloader = data.DataLoader(dataset, batch_size=128, num_workers=4, ↪
        ↪shuffle=False, drop_last=False)

        img_features = []
        img_targets = []
        for imgs, targets in tqdm(dataloader, "Extracting image features", leave=False):
            imgs = imgs.to(device)
            feats = model.model(imgs)
            img_features.append(feats.detach().cpu())
            img_targets.append(targets)
        img_features = torch.cat(img_features, dim=0)
        img_targets = torch.cat(img_targets, dim=0)
        # Sort by classes, so that we obtain tensors of shape [num_classes, exmps_per_
        ↪class, ...]
        # Makes it easier to process later
        img_targets, sort_idx = img_targets.sort()
        img_targets = img_targets.reshape(num_classes, exmps_per_class).transpose(0, 1)
        img_features = img_features[sort_idx].reshape(num_classes, exmps_per_class, -1).
        ↪transpose(0, 1)
```

(continues on next page)

(continued from previous page)

```

else:
    img_features, img_targets = data_feats

    # We iterate through the full dataset in two manners. First, to select the k-shot batch.
    # Second, to evaluate the model on all other examples
    accuracies = []
    for k_idx in tqdm(range(0, img_features.shape[0], k_shot), "Evaluating prototype classification", leave=False):
        # Select support set and calculate prototypes
        k_img_feats, k_targets = img_features[k_idx:k_idx+k_shot].flatten(0,1), img_targets[k_idx:k_idx+k_shot].flatten(0,1)
        prototypes, proto_classes = model.calculate_prototypes(k_img_feats, k_targets)
        # Evaluate accuracy on the rest of the dataset
        batch_acc = 0
        for e_idx in range(0, img_features.shape[0], k_shot):
            if k_idx == e_idx: # Do not evaluate on the support set examples
                continue
            e_img_feats, e_targets = img_features[e_idx:e_idx+k_shot].flatten(0,1), img_targets[e_idx:e_idx+k_shot].flatten(0,1)
            _, _, acc = model.classify_feats(prototypes, proto_classes, e_img_feats, e_targets)
            batch_acc += acc.item()
        batch_acc /= img_features.shape[0]//k_shot-1
        accuracies.append(batch_acc)

    return (mean(accuracies), stdev(accuracies)), (img_features, img_targets)

```

Testing ProtoNet is relatively quick if we have processed all images once. Hence, we can do in this notebook:

```

[21]: protonet_accuracies = dict()
data_feats = None
for k in [2, 4, 8, 16, 32]:
    protonet_accuracies[k], data_feats = test_proto_net(protonet_model, test_set, data_feats=data_feats, k_shot=k)
    print(f"Accuracy for k={k}: {100.0*protonet_accuracies[k][0]:.2f}% (+-{100*protonet_accuracies[k][1]:.2f}%)")

HBox(children=(HTML(value='Extracting image features'), FloatProgress(value=0.0, max=47.0), HTML(value='')))

/home/phillip/anaconda3/envs/dl2020/lib/python3.7/site-packages/torch/nn/functional.py:718: UserWarning: Named tensors and all their associated APIs are an experimental feature and subject to change. Please do not use them for anything important until they are released as stable. (Triggered internally at /opt/conda/conda-bld/pytorch-1623448265233/work/c10/core/TensorImpl.h:1156.)
    return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)

HBox(children=(HTML(value='Evaluating prototype classification'), FloatProgress(value=0.0, max=300.0), HTML(va...))

Accuracy for k=2: 44.30% (+-3.63%)

HBox(children=(HTML(value='Evaluating prototype classification'), FloatProgress(value=0.0, max=150.0), HTML(va...

```

```

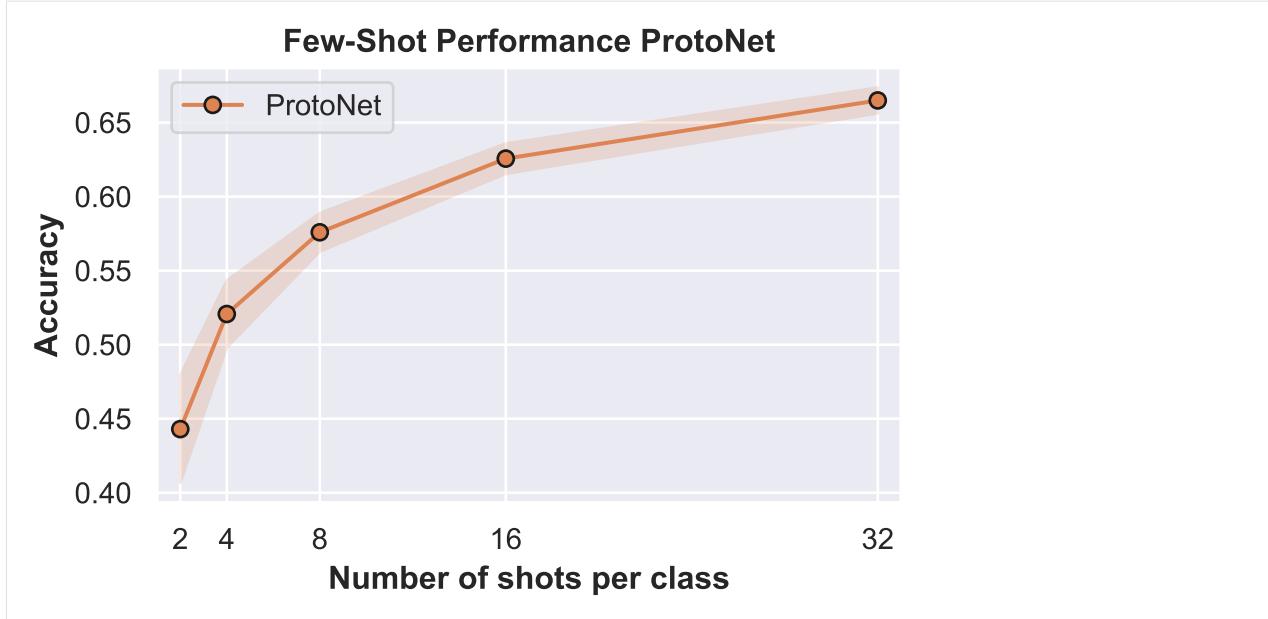
Accuracy for k=4: 52.07% (+-2.27%)
HBox(children=(HTML(value='Evaluating prototype classification'), FloatProgress(value=0.
˓→0, max=75.0), HTML(val...
Accuracy for k=8: 57.59% (+-1.30%)
HBox(children=(HTML(value='Evaluating prototype classification'), FloatProgress(value=0.
˓→0, max=38.0), HTML(val...
Accuracy for k=16: 62.56% (+-1.02%)
HBox(children=(HTML(value='Evaluating prototype classification'), FloatProgress(value=0.
˓→0, max=19.0), HTML(val...
Accuracy for k=32: 66.49% (+-0.87%)

```

Before discussing the results above, let's first plot the accuracies over number of examples in the support set:

```
[22]: def plot_few_shot(acc_dict, name, color=None, ax=None):
    sns.set()
    if ax is None:
        fig, ax = plt.subplots(1,1,figsize=(5,3))
    ks = sorted(list(acc_dict.keys()))
    mean_accs = [acc_dict[k][0] for k in ks]
    std_accs = [acc_dict[k][1] for k in ks]
    ax.plot(ks, mean_accs, marker='o', markeredgecolor='k', markersize=6, label=name, color=color)
    ax.fill_between(ks, [m-s for m,s in zip(mean_accs, std_accs)], [m+s for m,s in zip(mean_accs, std_accs)], alpha=0.2, color=color)
    ax.set_xticks(ks)
    ax.set_xlim([ks[0]-1, ks[-1]+1])
    ax.set_xlabel("Number of shots per class", weight='bold')
    ax.set_ylabel("Accuracy", weight='bold')
    if len(ax.get_title()) == 0:
        ax.set_title("Few-Shot Performance " + name, weight='bold')
    else:
        ax.set_title(ax.get_title() + " and " + name, weight='bold')
    ax.legend()
    return ax
```

```
[23]: ax = plot_few_shot(protonet_accuracies, name="ProtoNet", color="C1")
plt.show()
plt.close()
```



As we initially expected, the performance of ProtoNet indeed increases the more samples we have. However, even with just two samples per class, we classify almost half of the images correctly, which is well above random accuracy (10%). The curve shows an exponentially dampened trend, meaning that adding 2 extra examples to $k = 2$ has a much higher impact than adding 2 extra samples if we already have $k = 16$. Nonetheless, we can say that ProtoNet adapts fairly well to new classes.

4.17.3 MAML and ProtoMAML

The second meta-learning algorithm we will look at is MAML, short for Model-Agnostic Meta-Learning. MAML is an optimization-based meta-learning algorithm, which means that it tries to adjust the standard optimization procedure to a few-shot setting. The idea of MAML is relatively simple: given a model, support, and query set during training, we optimize the model for m steps on the support set and evaluate the gradients of the query loss with respect to the original model's parameters. For the same model, we do it for a few different support-query sets and accumulate the gradients. This results in learning a model that provides a good initialization for being quickly adapted to the training tasks. If we denote the model parameters with θ , we can visualize the procedure as follows (Figure credit - [Finn et al.](#)).

The full algorithm of MAML is therefore as follows. At each training step, we sample a batch of tasks, i.e., a batch of support-query set pairs. For each task T_i , we optimize a model f_θ on the support set via SGD, and denote this model as $f_{\theta'_i}$. We refer to this optimization as *inner loop*. Using this new model, we calculate the gradients of the original parameters, θ , with respect to the query loss on $f_{\theta'_i}$. These gradients are accumulated over all tasks and used to update θ . This is called *outer loop* since we iterate over tasks. The full MAML algorithm is summarized below (Figure credit - [Finn et al.](#)).

To obtain gradients for the initial parameters θ from the optimized model $f_{\theta'_i}$, we actually need second-order gradients, i.e. gradients of gradients, as the support set gradients depend on θ as well. This makes MAML computationally expensive, especially when using multiple inner loop steps. A simpler, yet almost equally well-performing alternative is First-Order MAML (FOMAML) which only uses first-order gradients. This means that the second-order gradients are ignored, and we can calculate the outer loop gradients (line 10 in algorithm 2) simply by calculating the gradients

with respect to θ'_i and use those as an update to θ . Hence, the new update rule becomes:

$$\theta \leftarrow \theta - \beta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \nabla_{\theta'_i} \mathcal{L}_{\mathcal{T}_i}(f_{\theta'_i})$$

Note the change of θ to θ'_i for ∇ .

ProtoMAML

A problem of MAML is how to design the output classification layer. In case all tasks have a different number of classes, we need to initialize the output layer with zeros or randomly in every iteration. Even if we always have the same number of classes, we just start from random predictions. This requires several inner loop steps to reach a reasonable classification result. To overcome this problem, Triantafillou et al. (2020) propose to combine the merits of Prototypical Networks and MAML. Specifically, we can use prototypes to initialize our output layer to have a strong initialization. Thereby, it can be shown that the softmax over euclidean distances can be reformulated as a linear layer with softmax. To see this, let's first write out the negative Euclidean distance between a feature vector $f_\theta(\mathbf{x}^*)$ of a new data point \mathbf{x}^* to a prototype \mathbf{v}_c of class c :

$$-\|f_\theta(\mathbf{x}^*) - \mathbf{v}_c\|^2 = -f_\theta(\mathbf{x}^*)^T f_\theta(\mathbf{x}^*) + 2\mathbf{v}_c^T f_\theta(\mathbf{x}^*) - \mathbf{v}_c^T \mathbf{v}_c$$

We perform the classification across all classes $c \in \mathcal{C}$ and take a softmax on the distance. Hence, any term that is the same for all classes can be removed without changing the output probabilities. In the equation above, this is true for $-f_\theta(\mathbf{x}^*)^T f_\theta(\mathbf{x}^*)$ since it is independent of any class prototype. Thus, we can write:

$$-\|f_\theta(\mathbf{x}^*) - \mathbf{v}_c\|^2 = 2\mathbf{v}_c^T f_\theta(\mathbf{x}^*) - \|\mathbf{v}_c\|^2 + \text{constant}$$

Taking a second look at the equation above, it looks a lot like a linear layer. For this, we use $\mathbf{W}_{c,\cdot} = 2\mathbf{v}_c$ and $b_c = -\|\mathbf{v}_c\|^2$ which gives us the linear layer $\mathbf{W}f_\theta(\mathbf{x}^*) + \mathbf{b}$. Hence, if we initialize the output weight with twice the prototypes, and the biases by the negative squared L2 norm of the prototypes, we start with a Prototypical Network. MAML allows us to adapt this layer and the rest of the network further.

In the following, we will implement First-Order ProtoMAML for few-shot classification. The implementation of MAML would be the same except for the output layer initialization.

ProtoMAML implementation

For implementing ProtoMAML, we can follow Algorithm 2 with minor modifications. At each training step, we first sample a batch of tasks, and a support and query set for each task. In our case of few-shot classification, this means that we simply sample multiple support-query set pairs from our sampler. For each task, we finetune our current model on the support set. However, since we need to remember the original parameters for the other tasks, the outer loop gradient update, and future training steps, we need to create a copy of our model and finetune only the copy. We can copy a model by using standard Python functions like `deepcopy`. The inner loop is implemented in the function `adapt_few_shot` in the PyTorch Lightning module below.

After finetuning the model, we apply it to the query set and calculate the first-order gradients with respect to the original parameters θ . In contrast to simple MAML, we also have to consider the gradients with respect to the output layer initialization, i.e. the prototypes, since they directly rely on θ . To realize this efficiently, we take two steps. First, we calculate the prototypes by applying the original model, i.e. not the copied model, on the support elements. When initializing the output layer, we detach the prototypes to stop the gradients. This is because, in the inner loop itself, we do not want to consider gradients through the prototypes back to the original model. However, after the inner loop is finished, we re-attach the computation graph of the prototypes by writing `output_weight = (output_weight - init_weight).detach() + init_weight`. While this line does not change the value of the variable `output_weight`, it adds its dependency on the prototype initialization `init_weight`. Thus, if we call `.backward` on `output_weight`, we will automatically calculate the first-order gradients with respect to the prototype initialization in the original model.

After calculating all gradients and summing them together in the original model, we can take a standard optimizer step. PyTorch Lightning's method is however designed to return a loss-tensor on which we call `.backward` first. Since this is not possible here, we need to perform the optimization step ourselves. All details can be found in the code below.

For implementing (Proto-)MAML with second-order gradients, it is recommended to use libraries such as `:math:nabla`higher <https://github.com/facebookresearch/higher>`` from Facebook AI Research. For simplicity, we stick with first-order methods here.

```
[24]: class ProtoMAML(pl.LightningModule):

    def __init__(self, proto_dim, lr, lr_inner, lr_output, num_inner_steps):
        """
        Inputs
        proto_dim - Dimensionality of prototype feature space
        lr - Learning rate of the outer loop Adam optimizer
        lr_inner - Learning rate of the inner loop SGD optimizer
        lr_output - Learning rate for the output layer in the inner loop
        num_inner_steps - Number of inner loop updates to perform
        """
        super().__init__()
        self.save_hyperparameters()
        self.model = get_convnet(output_size=self.hparams.proto_dim)

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(), lr=self.hparams.lr)
        scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[140, 180],  
→gamma=0.1)
        return [optimizer], [scheduler]

    def run_model(self, local_model, output_weight, output_bias, imgs, labels):
        # Execute a model with given output layer weights and inputs
        feats = local_model(imgs)
        preds = F.linear(feats, output_weight, output_bias)
        loss = F.cross_entropy(preds, labels)
        acc = (preds.argmax(dim=1) == labels).float()
        return loss, preds, acc

    def adapt_few_shot(self, support_imgs, support_targets):
        # Determine prototype initialization
        support_feats = self.model(support_imgs)
        prototypes, classes = ProtoNet.calculate_prototypes(support_feats, support_
→targets)
        support_labels = (classes[None, :] == support_targets[:, None]).long().argmax(dim=-  
→1)
        # Create inner-loop model and optimizer
        local_model = deepcopy(self.model)
        local_model.train()
        local_optim = optim.SGD(local_model.parameters(), lr=self.hparams.lr_inner)
        local_optim.zero_grad()
        # Create output layer weights with prototype-based initialization
        init_weight = 2 * prototypes
        init_bias = -torch.norm(prototypes, dim=1)**2
        output_weight = init_weight.detach().requires_grad_()
        output_bias = init_bias.detach().requires_grad_()
```

(continues on next page)

(continued from previous page)

```

# Optimize inner loop model on support set
for _ in range(self.hparams.num_inner_steps):
    # Determine loss on the support set
    loss, _, _ = self.run_model(local_model, output_weight, output_bias, support_
 imgs, support_labels)
    # Calculate gradients and perform inner loop update
    loss.backward()
    local_optim.step()
    # Update output layer via SGD
    output_weight.data -= self.hparams.lr_output * output_weight.grad
    output_bias.data -= self.hparams.lr_output * output_bias.grad
    # Reset gradients
    local_optim.zero_grad()
    output_weight.grad.fill_(0)
    output_bias.grad.fill_(0)

    # Re-attach computation graph of prototypes
    output_weight = (output_weight - init_weight).detach() + init_weight
    output_bias = (output_bias - init_bias).detach() + init_bias

return local_model, output_weight, output_bias, classes

def outer_loop(self, batch, mode="train"):
    accuracies = []
    losses = []
    self.model.zero_grad()

    # Determine gradients for batch of tasks
    for task_batch in batch:
        imgs, targets = task_batch
        support_imgs, query_imgs, support_targets, query_targets = split_batch(imgs,_
 targets)
        # Perform inner loop adaptation
        local_model, output_weight, output_bias, classes = self.adapt_few_-
 shot(support_imgs, support_targets)
        # Determine loss of query set
        query_labels = (classes[None,:,:] == query_targets[:,None]).long().argmax(dim=-
 1)
        loss, preds, acc = self.run_model(local_model, output_weight, output_bias,_
 query_imgs, query_labels)
        # Calculate gradients for query set loss
        if mode == "train":
            loss.backward()

            for p_global, p_local in zip(self.model.parameters(), local_model.-
 parameters()):
                p_global.grad += p_local.grad # First-order approx. -> add_
 gradients of finetuned and base model

        accuracies.append(acc.mean().detach())
        losses.append(loss.detach())

```

(continues on next page)

(continued from previous page)

```

# Perform update of base model
if mode == "train":
    opt = self.optimizers()
    opt.step()
    opt.zero_grad()

    self.log(f"{mode}_loss", sum(losses) / len(losses))
    self.log(f"{mode}_acc", sum(accuracies) / len(accuracies))

def training_step(self, batch, batch_idx):
    self.outer_loop(batch, mode="train")
    return None # Returning None means we skip the default training optimizer steps,
    ↪ by PyTorch Lightning

def validation_step(self, batch, batch_idx):
    # Validation requires to finetune a model, hence we need to enable gradients
    torch.set_grad_enabled(True)
    self.outer_loop(batch, mode="val")
    torch.set_grad_enabled(False)

```

Training

To train ProtoMAML, we need to change our sampling slightly. Instead of a single support-query set batch, we need to sample multiple. To implement this, we yet use another Sampler that combines multiple batches from a FewShotBatchSampler and returns it afterward. Additionally, we define a `collate_fn` for our data loader which takes the stack of support-query set images and returns the tasks as a list. This makes it easier to process in our PyTorch Lightning module before. The implementation of the sampler can be found below.

```
[25]: class TaskBatchSampler(object):

    def __init__(self, dataset_targets, batch_size, N_way, K_shot, include_query=False,
    ↪ shuffle=True):
        """
        Inputs:
            dataset_targets - PyTorch tensor of the labels of the data elements.
            batch_size - Number of tasks to aggregate in a batch
            N_way - Number of classes to sample per batch.
            K_shot - Number of examples to sample per class in the batch.
            include_query - If True, returns batch of size N_way*K_shot*2, which
                            can be split into support and query set. Simplifies
                            the implementation of sampling the same classes but
                            distinct examples for support and query set.
            shuffle - If True, examples and classes are newly shuffled in each
                      iteration (for training)
        """
        super().__init__()
        self.batch_sampler = FewShotBatchSampler(dataset_targets, N_way, K_shot, include_
        ↪ query, shuffle)
        self.task_batch_size = batch_size
        self.local_batch_size = self.batch_sampler.batch_size
```

(continues on next page)

(continued from previous page)

```

def __iter__(self):
    # Aggregate multiple batches before returning the indices
    batch_list = []
    for batch_idx, batch in enumerate(self.batch_sampler):
        batch_list.extend(batch)
        if (batch_idx+1) % self.task_batch_size == 0:
            yield batch_list
            batch_list = []

def __len__(self):
    return len(self.batch_sampler)//self.task_batch_size

def get_collate_fn(self):
    # Returns a collate function that converts one big tensor into a list of task-
    ↪specific tensors
    def collate_fn(item_list):
        imgs = torch.stack([img for img, target in item_list], dim=0)
        targets = torch.stack([target for img, target in item_list], dim=0)
        imgs = imgs.chunk(self.task_batch_size, dim=0)
        targets = targets.chunk(self.task_batch_size, dim=0)
        return list(zip(imgs, targets))
    return collate_fn

```

The creation of the data loaders is with this sampler straight-forward. Note that since many images need to be loaded for a training batch, it is recommended to use less workers than usual.

```

[26]: # Training constant (same as for ProtoNet)
N_WAY = 5
K_SHOT = 4

# Training set
train_protomaml_sampler = TaskBatchSampler(train_set.targets,
                                             include_query=True,
                                             N_way=N_WAY,
                                             K_shot=K_SHOT,
                                             batch_size=16)
train_protomaml_loader = data.DataLoader(train_set,
                                         batch_sampler=train_protomaml_sampler,
                                         collate_fn=train_protomaml_sampler.get_collate_
                                         ↪fn(),
                                         num_workers=2)

# Validation set
val_protomaml_sampler = TaskBatchSampler(val_set.targets,
                                           include_query=True,
                                           N_way=N_WAY,
                                           K_shot=K_SHOT,
                                           batch_size=1, # We do not update the
                                           ↪parameters, hence the batch size is irrelevant here
                                           shuffle=False)
val_protomaml_loader = data.DataLoader(val_set,

```

(continues on next page)

(continued from previous page)

```
batch_sampler=val_protomaml_sampler,  
collate_fn=val_protomaml_sampler.get_collate_fn(),  
num_workers=2)
```

Now, we are ready to train our ProtoMAML. We use the same feature space size as for ProtoNet but can use a higher learning rate since the outer loop gradients are accumulated over 16 batches. The inner loop learning rate is set to 0.1, which is much higher than the outer loop learning rate because we use SGD in the inner loop instead of Adam. Commonly, the learning rate for the output layer is higher than the base model if the base model is very deep or pre-trained. However, for our setup, we observed no noticeable impact of using a different learning rate than the base model. The number of inner loop updates is another crucial hyperparameter and depends on the similarity of our training tasks. Since all tasks are on images from the same dataset, we notice that a single inner loop update achieves similar performance as 3 or 5 while training considerably faster. However, especially in RL and NLP, a larger number of inner loop steps are often needed.

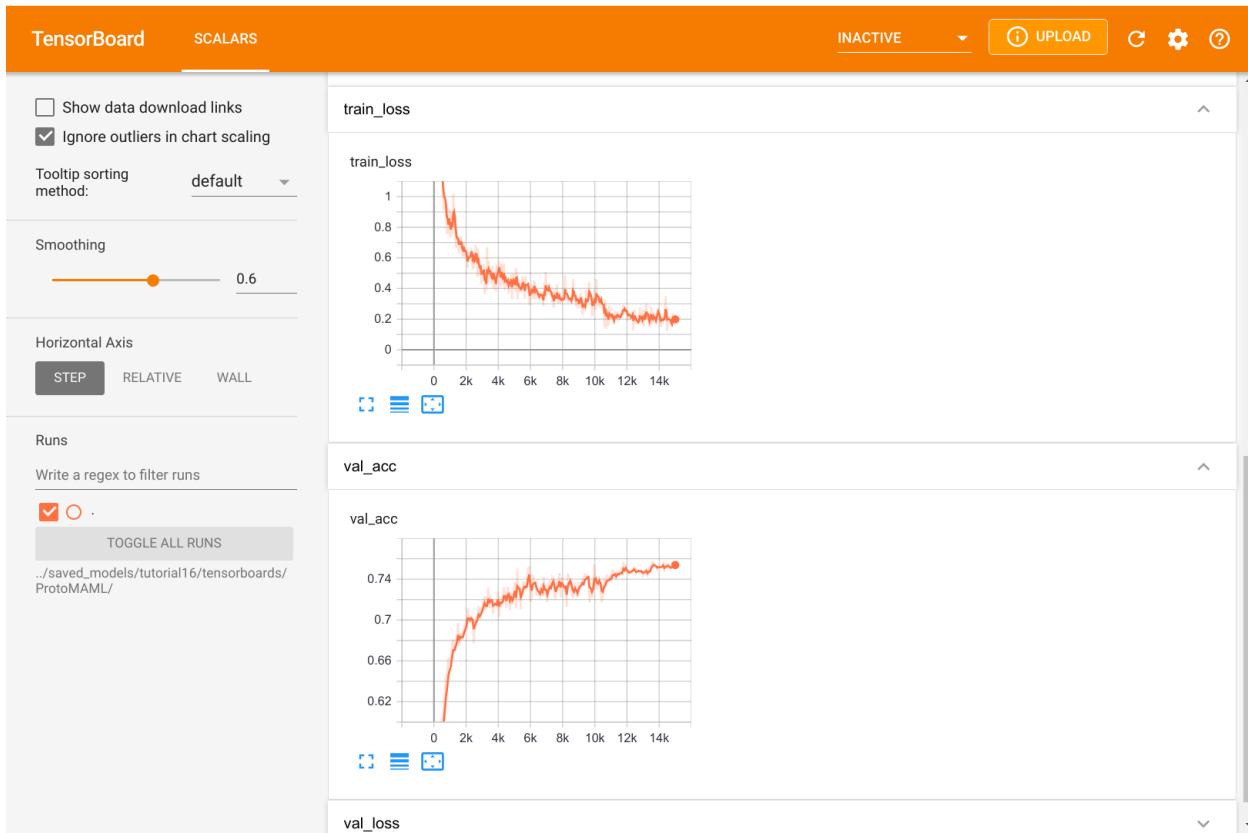
```
[27]: protomaml_model = train_model(ProtoMAML,  
                                  proto_dim=64,  
                                  lr=1e-3,  
                                  lr_inner=0.1,  
                                  lr_output=0.1,  
                                  num_inner_steps=1, # Often values between 1 and 10  
                                  train_loader=train_protomaml_loader,  
                                  val_loader=val_protomaml_loader)
```

```
GPU available: True, used: True  
TPU available: False, using: 0 TPU cores
```

```
Found pretrained model at ../saved_models/tutorial16/ProtoMAML.ckpt, loading...
```

Let's have a look at the training TensorBoard.

```
[28]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH if needed  
%tensorboard --logdir ../saved_models/tutorial16/tensorboards/ProtoMAML/
```



One obvious difference to ProtoNet is that the loss curves look much less noisy. This is because we average the outer loop gradients over multiple tasks, and thus have a smoother training curve. Additionally, we only have 15k training iterations after 200 epochs. This is again because of the task batches, which cause 16 times fewer iterations. However, each iteration has seen 16 times more data in this experiment. Thus, we still have a fair comparison between ProtoMAML and ProtoNet. At first sight on the validation accuracy, one would assume that ProtoNet performs superior to ProtoMAML, but we have to verify that with proper testing below.

Testing

We test ProtoMAML in the same manner as ProtoNet, namely by picking random examples in the test set as support sets and use the rest of the dataset as the query set. Instead of just calculating the prototypes for all examples, we need to finetune a separate model for each support set. This is why this process is more expensive than ProtoNet, and in our case, testing $k = \{2, 4, 8, 16, 32\}$ can take almost an hour. Hence, we provide evaluation files besides the pretrained models.

```
[29]: def test_protomaml(model, dataset, k_shot=4):
    pl.seed_everything(42)
    model = model.to(device)
    num_classes = dataset.targets.unique().shape[0]
    exmps_per_class = dataset.targets.shape[0]//num_classes

    # Data loader for full test set as query set
    full_dataloader = data.DataLoader(dataset,
                                      batch_size=128,
                                      num_workers=4,
                                      shuffle=False,
```

(continues on next page)

(continued from previous page)

```

        drop_last=False)

# Data loader for sampling support sets
sampler = FewShotBatchSampler(dataset.targets,
                             include_query=False,
                             N_way=num_classes,
                             K_shot=k_shot,
                             shuffle=False,
                             shuffle_once=False)
sample_dataloader = data.DataLoader(dataset,
                                    batch_sampler=sampler,
                                    num_workers=2)

# We iterate through the full dataset in two manners. First, to select the k-shot,
batch.

# Second, the evaluate the model on all other examples
accuracies = []
for (support_imgs, support_targets), support_indices in tqdm(zip(sample_dataloader, sampler),
    "Performing few-shot finetuning"):
    support_imgs = support_imgs.to(device)
    support_targets = support_targets.to(device)
    # Finetune new model on support set
    local_model, output_weight, output_bias, classes = model.adapt_few_shot(support_
    imgs, support_targets)
    with torch.no_grad(): # No gradients for query set needed
        local_model.eval()
        batch_acc = torch.zeros((0,), dtype=torch.float32, device=device)
        # Evaluate all examples in test dataset
        for query_imgs, query_targets in full_dataloader:
            query_imgs = query_imgs.to(device)
            query_targets = query_targets.to(device)
            query_labels = (classes[None,:,:] == query_targets[:,None]).long()
            argmax(dim=-1)
            _, _, acc = model.run_model(local_model, output_weight, output_bias,
            query_imgs, query_labels)
            batch_acc = torch.cat([batch_acc, acc.detach()], dim=0)
        # Exclude support set elements
        for s_idx in support_indices:
            batch_acc[s_idx] = 0
        batch_acc = batch_acc.sum().item() / (batch_acc.shape[0] - len(support_
        indices))
        accuracies.append(batch_acc)
return mean(accuracies), stdev(accuracies)

```

In contrast to training, it is recommended to use many more inner loop updates during testing. During training, we are not interested in getting the best model from the inner loop, but the model which can provide the best gradients. Hence, one update might be already sufficient in training, but for testing, it was often observed that a larger number of updates can give a considerable performance boost. Thus, we change the inner loop updates to 200 before testing.

[30]: protomaml_model.hparams.num_inner_steps = 200

Now, we can test our model. For the pre-trained models, we provide a json file with the results to reduce evaluation time.

```
[31]: protomaml_result_file = os.path.join(CHECKPOINT_PATH, "protomaml_fewshot.json")

if os.path.isfile(protomaml_result_file):
    # Load pre-computed results
    with open(protomaml_result_file, 'r') as f:
        protomaml_accuracies = json.load(f)
    protomaml_accuracies = {int(k): v for k, v in protomaml_accuracies.items()}
else:
    # Perform same experiments as for ProtoNet
    protomaml_accuracies = dict()
    for k in [2, 4, 8, 16, 32]:
        protomaml_accuracies[k] = test_protomaml(protomaml_model, test_set, k_shot=k)
    # Export results
    with open(protomaml_result_file, 'w') as f:
        json.dump(protomaml_accuracies, f, indent=4)

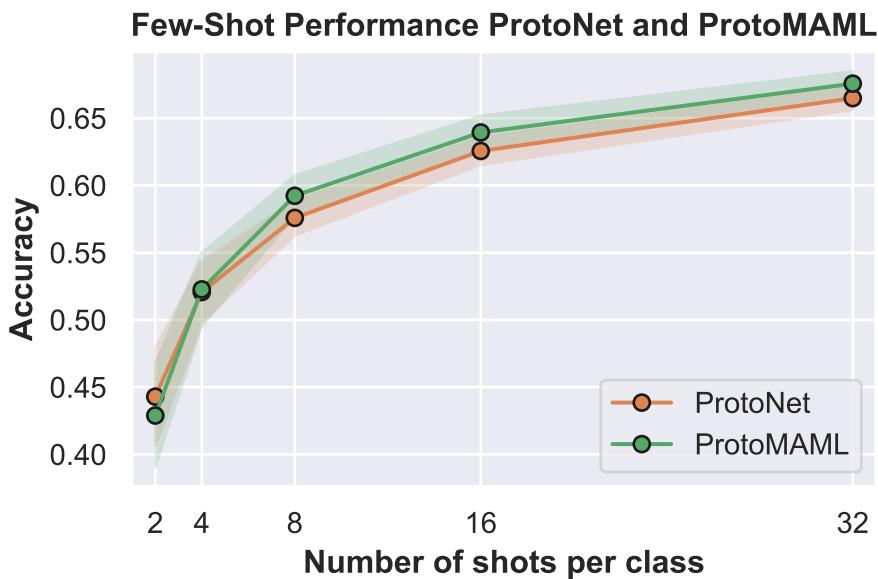
for k in protomaml_accuracies:
    print(f"Accuracy for k={k}: {100.0*protomaml_accuracies[k][0]:.2f}% (+-{100.0*protomaml_accuracies[k][1]:.2f}%)")

```

Accuracy for k=2: 42.89% (+-3.82%)
 Accuracy for k=4: 52.27% (+-2.72%)
 Accuracy for k=8: 59.23% (+-1.50%)
 Accuracy for k=16: 63.94% (+-1.24%)
 Accuracy for k=32: 67.57% (+-0.90%)

Again, let's plot the results in our plot from before.

```
[32]: ax = plot_few_shot(protonet_accuracies, name="ProtoNet", color="C1")
plot_few_shot(protomaml_accuracies, name="ProtoMAML", color="C2", ax=ax)
plt.show()
plt.close()
```



We can observe that ProtoMAML is indeed able to outperform ProtoNet for $k > 4$. This is because, with more samples, it becomes more relevant to also adapt the base model's parameters. Meanwhile, for $k = 2$, ProtoMAML

achieves lower performance than ProtoNet. This is likely also related to choosing 200 inner loop updates since with more updates, there exists the risk of overfitting. Nonetheless, the high standard deviation for $k = 2$ makes it hard to take any statistically valid conclusion.

Overall, we can conclude that ProtoMAML slightly outperforms ProtoNet for larger shot counts. However, one disadvantage of ProtoMAML is its much longer training and testing time. ProtoNet provides a simple, efficient, yet strong baseline for ProtoMAML, and might be the better solution in situations where limited resources are available.

4.17.4 Domain adaptation

So far, we have evaluated our meta-learning algorithms on the same dataset on which we have trained them. However, meta-learning algorithms are especially interesting when we want to move from one to another dataset. So, what happens if we apply them on a quite different dataset than CIFAR? This is what we try out below, and evaluate ProtoNet and ProtoMAML on the SVHN dataset.

SVHN dataset

The Street View House Numbers (SVHN) dataset is a real-world image dataset for house number detection. It is similar to MNIST by having the classes 0 to 9, but is more difficult due to its real-world setting and possible distracting numbers left and right. Let's first load the dataset, and visualize some images to get an impression of the dataset.

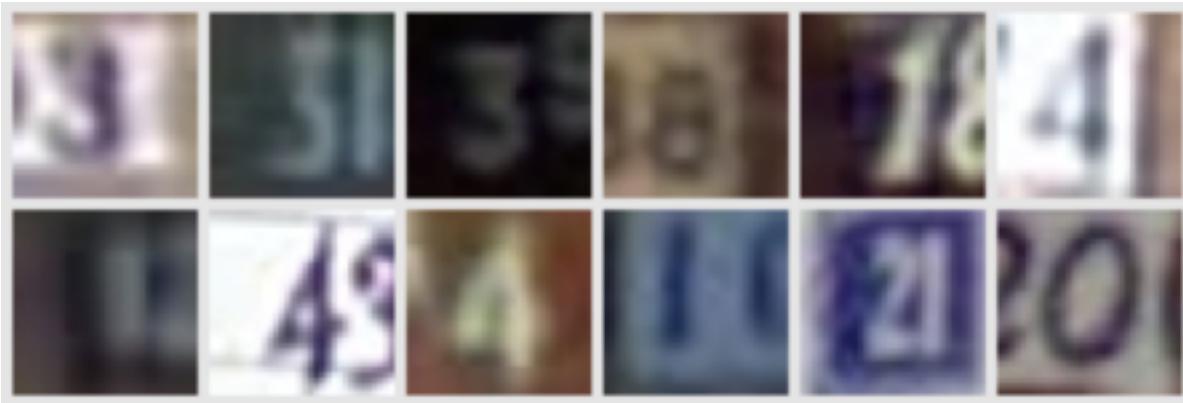
```
[33]: SVHN_test_dataset = SVHN(root=DATASET_PATH, split='test', download=True,
    ↪transform=transforms.ToTensor())
```

```
Using downloaded and verified file: ../data/test_32x32.mat
```

```
[34]: # Visualize some examples
NUM_IMAGES = 12
SVHN_images = torch.stack([SVHN_test_dataset[np.random.randint(len(SVHN_test_
    ↪dataset))][0] for idx in range(NUM_IMAGES)], dim=0)
img_grid = torchvision.utils.make_grid(SVHN_images, nrow=6, normalize=True, pad_value=0.
    ↪9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("Image examples of the SVHN dataset")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

Image examples of the SVHN dataset



Each image is labeled with one class between 0 and 9 representing the main digit in the image. Can our ProtoNet and ProtoMAML learn to classify the digits from only a few examples? This is what we will test out below. The images have the same size as CIFAR, so that we can use the images without changes. We first prepare the dataset, for which we take the first 500 images per class. For this dataset, we use our test functions as before to get an estimated performance for different number of shots.

```
[35]: imgs = np.transpose(SVHN_test_dataset.data, (0, 2, 3, 1))
targets = SVHN_test_dataset.labels
min_label_count = min(500, np.bincount(SVHN_test_dataset.labels).min()) # Limit number
# of examples to 500 to reduce test time

idxs = np.concatenate([np.where(targets==c)[0][:min_label_count] for c in
range(1+targets.max())], axis=0)
imgs = imgs[idxs]
targets = torch.from_numpy(targets[idxs]).long()

svhn_fewshot_dataset = ImageDataset(imgs, targets, img_transform=test_transform)
svhn_fewshot_dataset.imgs.shape
```

[35]: (5000, 32, 32, 3)

Experiments

First, we can apply ProtoNet to the SVHN dataset:

```
[36]: protonet_svhn_accuracies = dict()
data_feats = None
for k in [2, 4, 8, 16, 32]:
    protonet_svhn_accuracies[k], data_feats = test_proto_net(protonet_model, svhn_
#fewshot_dataset, data_feats=data_feats, k_shot=k)
    print(f"Accuracy for k={k}: {100.0*protonet_svhn_accuracies[k][0]:.2f}% (+-
#{100*protonet_svhn_accuracies[k][1]:.2f}%)")

HBox(children=(HTML(value='Extracting image features'), FloatProgress(value=0.0, max=40.
#0), HTML(value='')))

HBox(children=(HTML(value='Evaluating prototype classification'), FloatProgress(value=0.
#0, max=250.0), HTML(va...))
```

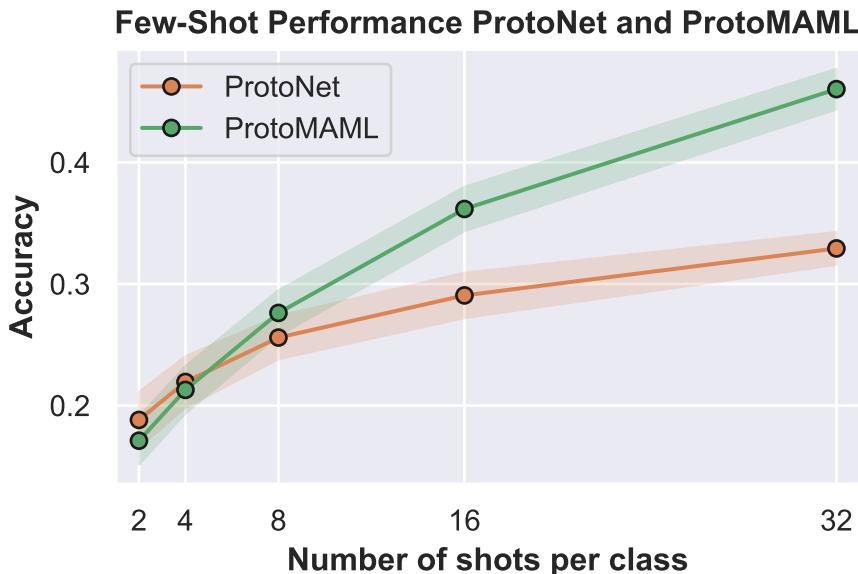
```
Accuracy for k=2: 18.82% (+-2.28%)  
HBox(children=(HTML(value='Evaluating prototype classification'), FloatProgress(value=0.  
˓→0, max=125.0), HTML(va...  
Accuracy for k=4: 21.94% (+-2.09%)  
HBox(children=(HTML(value='Evaluating prototype classification'), FloatProgress(value=0.  
˓→0, max=63.0), HTML(val...  
Accuracy for k=8: 25.59% (+-1.76%)  
HBox(children=(HTML(value='Evaluating prototype classification'), FloatProgress(value=0.  
˓→0, max=32.0), HTML(val...  
Accuracy for k=16: 29.06% (+-1.84%)  
HBox(children=(HTML(value='Evaluating prototype classification'), FloatProgress(value=0.  
˓→0, max=16.0), HTML(val...  
Accuracy for k=32: 32.93% (+-1.33%)
```

It becomes clear that the results are much lower than the ones on CIFAR, and just slightly above random for $k = 2$. How about ProtoMAML? We provide again evaluation files since the evaluation can take several minutes to complete.

```
[37]: protomaml_result_file = os.path.join(CHECKPOINT_PATH, "protomaml_svhn_fewshot.json")  
  
if os.path.isfile(protomaml_result_file):  
    # Load pre-computed results  
    with open(protomaml_result_file, 'r') as f:  
        protomaml_svhn_accuracies = json.load(f)  
    protomaml_svhn_accuracies = {int(k): v for k, v in protomaml_svhn_accuracies.items()}  
else:  
    # Perform same experiments as for ProtoNet  
    protomaml_svhn_accuracies = dict()  
    for k in [2, 4, 8, 16, 32]:  
        protomaml_svhn_accuracies[k] = test_protomaml(protomaml_model, svhn_fewshot_  
˓→dataset, k_shot=k)  
    # Export results  
    with open(protomaml_result_file, 'w') as f:  
        json.dump(protomaml_svhn_accuracies, f, indent=4)  
  
for k in protomaml_svhn_accuracies:  
    print(f"Accuracy for k={k}: {100.0*protomaml_svhn_accuracies[k][0]:.2f}% (+-{100.  
˓→0*protomaml_svhn_accuracies[k][1]:.2f}%)")  
  
Accuracy for k=2: 17.11% (+-1.95%)  
Accuracy for k=4: 21.29% (+-1.92%)  
Accuracy for k=8: 27.62% (+-1.84%)  
Accuracy for k=16: 36.17% (+-1.80%)  
Accuracy for k=32: 46.03% (+-1.65%)
```

While ProtoMAML shows similar performance than ProtoNet for $k \leq 4$, it considerably outperforms ProtoNet for more than 8 shots. This is because we can adapt the base model, which is crucial when the data does not fit the original training data. For $k = 32$, ProtoMAML achieves 13% higher classification accuracy than ProtoNet which already starts to flatten out. We can see the trend more clearly in our plot below.

```
[38]: ax = plot_few_shot(protonet_svhn_accuracies, name="ProtoNet", color="C1")
plot_few_shot(protomaml_svhn_accuracies, name="ProtoMAML", color="C2", ax=ax)
plt.show()
plt.close()
```



4.17.5 Conclusion

In this notebook, we have discussed meta-learning algorithms that learn to adapt to new classes and/or tasks with just a few samples. We have discussed three popular algorithms, namely ProtoNet, MAML, and ProtoMAML. On the few-shot image classification task of CIFAR100, ProtoNet and ProtoMAML showed to perform similarly well, with slight benefits of ProtoMAML for larger shot sizes. However, for out-of-distribution data (SVHN), the ability to optimize the base model showed to be crucial and gave ProtoMAML considerable performance gains over ProtoNet. Nonetheless, ProtoNet offers other advantages compared to ProtoMAML, namely a very cheap training and test cost as well as a simpler implementation. Hence, it is recommended to consider whether the additional complexity of ProtoMAML is worth the extra training computation cost, or whether ProtoNet is already sufficient for the task at hand.

References

- [1] Snell, Jake, Kevin Swersky, and Richard S. Zemel. “Prototypical networks for few-shot learning.” NeurIPS 2017. ([link](#))
- [2] Chelsea Finn, Pieter Abbeel, Sergey Levine. “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks.” ICML 2017. ([link](#))
- [3] Triantafillou, Eleni, Tyler Zhu, Vincent Dumoulin, Pascal Lamblin, Utku Evci, Kelvin Xu, Ross Goroshin et al. “Meta-dataset: A dataset of datasets for learning to learn from few examples.” ICLR 2020. ([link](#))

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.18 Tutorial 17: Self-Supervised Contrastive Learning with SimCLR

Filled notebook:

Pre-trained models:

Recordings:

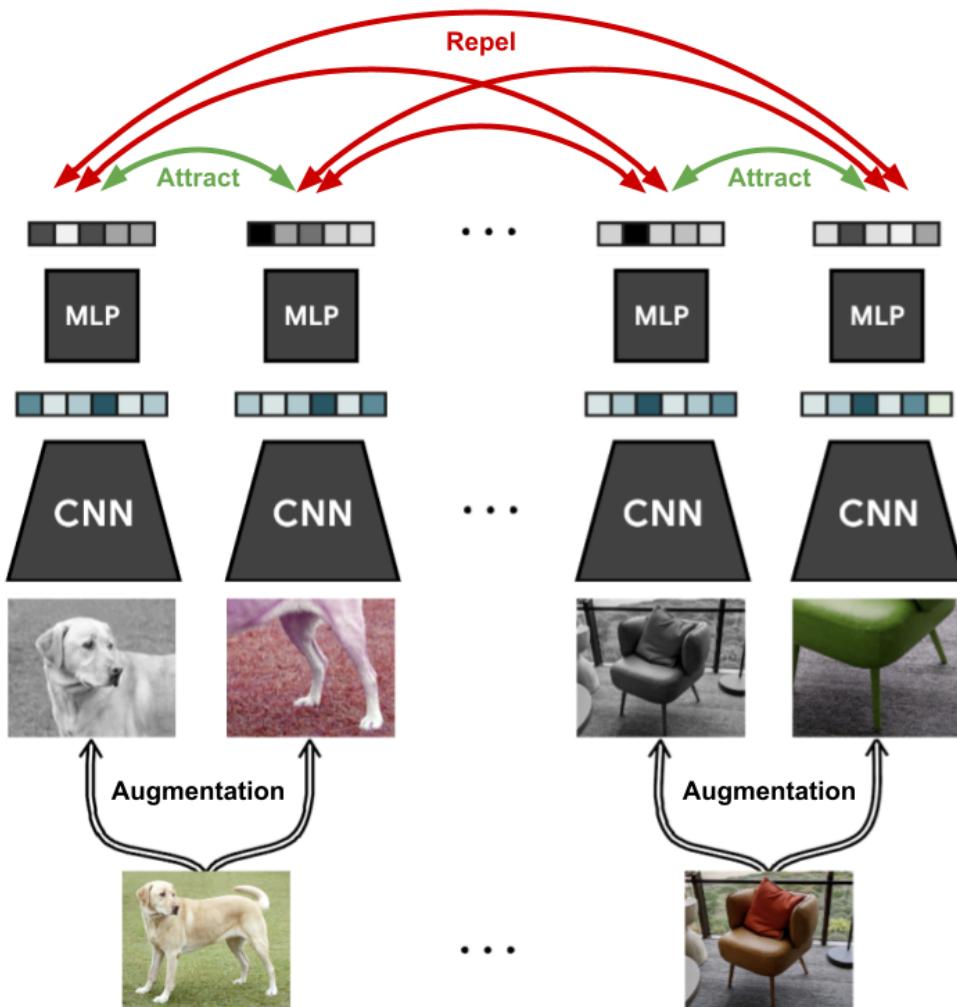
JAX+Flax version:

Author: Phillip Lippe

Note: Interested in JAX? Check out our [JAX+Flax version](#) of this tutorial!

In this tutorial, we will take a closer look at self-supervised contrastive learning. Self-supervised learning, or also sometimes called unsupervised learning, describes the scenario where we have given input data, but no accompanying labels to train in a classical supervised way. However, this data still contains a lot of information from which we can learn: how are the images different from each other? What patterns are descriptive for certain images? Can we cluster the images? And so on. Methods for self-supervised learning try to learn as much as possible from the data alone, so it can quickly be finetuned for a specific classification task. The benefit of self-supervised learning is that a large dataset can often easily be obtained. For instance, if we want to train a vision model on semantic segmentation for autonomous driving, we can collect large amounts of data by simply installing a camera in a car, and driving through a city for an hour. In contrast, if we would want to do supervised learning, we would have to manually label all those images before training a model. This is extremely expensive, and would likely take a couple of months to manually label the same amount of data. Further, self-supervised learning can provide an alternative to transfer learning from models pretrained on ImageNet since we could pretrain a model on a specific dataset/situation, e.g. traffic scenarios for autonomous driving.

Within the last two years, a lot of new approaches have been proposed for self-supervised learning, in particular for images, that have resulted in great improvements over supervised models when few labels are available. The subfield that we will focus on in this tutorial is contrastive learning. Contrastive learning is motivated by the question mentioned above: how are images different from each other? Specifically, contrastive learning methods train a model to cluster an image and its slightly augmented version in latent space, while the distance to other images should be maximized. A very recent and simple method for this is [SimCLR](#), which is visualized below (figure credit - [Ting Chen et al.](#)).



The general setup is that we are given a dataset of images without any labels, and want to train a model on this data such that it can quickly adapt to any image recognition task afterward. During each training iteration, we sample a batch of images as usual. For each image, we create two versions by applying data augmentation techniques like cropping, Gaussian noise, blurring, etc. An example of such is shown on the left with the image of the dog. We will go into the details and effects of the chosen augmentation techniques later. On those images, we apply a CNN like ResNet and obtain as output a 1D feature vector on which we apply a small MLP. The output features of the two augmented images are then trained to be close to each other, while all other images in that batch should be as different as possible. This way, the model has to learn to recognize the content of the image that remains unchanged under the data augmentations, such as objects which we usually care about in supervised tasks.

We will now implement this framework ourselves and discuss further details along the way. Let's first start with importing our standard libraries below:

```
[1]: ## Standard libraries
import os
from copy import deepcopy

## Imports for plotting
import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
```

(continues on next page)

(continued from previous page)

```
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.set()

## tqdm for loading bars
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim

## Torchvision
import torchvision
from torchvision.datasets import STL10
from torchvision import transforms

# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Import tensorboard
%load_ext tensorboard

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial17"
# In this notebook, we use data loaders with heavier computational processing. It is recommended to use as many
# workers as possible in a data loader, which corresponds to the number of CPU cores
NUM_WORKERS = os.cpu_count()

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", device)
```

(continues on next page)

(continued from previous page)

```
print("Number of workers:", NUM_WORKERS)
Global seed set to 42
Device: cuda:0
Number of workers: 16
```

As in many tutorials before, we provide pre-trained models. Note that those models are slightly larger as normal (~100MB overall) since we use the default ResNet-18 architecture. If you are running this notebook locally, make sure to have sufficient disk space available.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial17/"
# Files to download
pretrained_files = ["SimCLR.ckpt", "ResNet.ckpt",
                    "tensorboards/SimCLR/events.out.tfevents.SimCLR",
                    "tensorboards/classification/ResNet/events.out.tfevents.ResNet"]
pretrained_files += [f"LogisticRegression_{size}.ckpt" for size in [10, 20, 50, 100, 200,
                                                               500]]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive folder, or contact the author with the full output including the following error:\n", e)
```

4.18.1 SimCLR

We will start our exploration of contrastive learning by discussing the effect of different data augmentation techniques, and how we can implement an efficient data loader for such. Next, we implement SimCLR with PyTorch Lightning, and finally train it on a large, unlabeled dataset.

Data Augmentation for Contrastive Learning

To allow efficient training, we need to prepare the data loading such that we sample two different, random augmentations for each image in the batch. The easiest way to do this is by creating a transformation that, when being called, applies a set of data augmentations to an image twice. This is implemented in the class `ContrastiveTransformations` below:

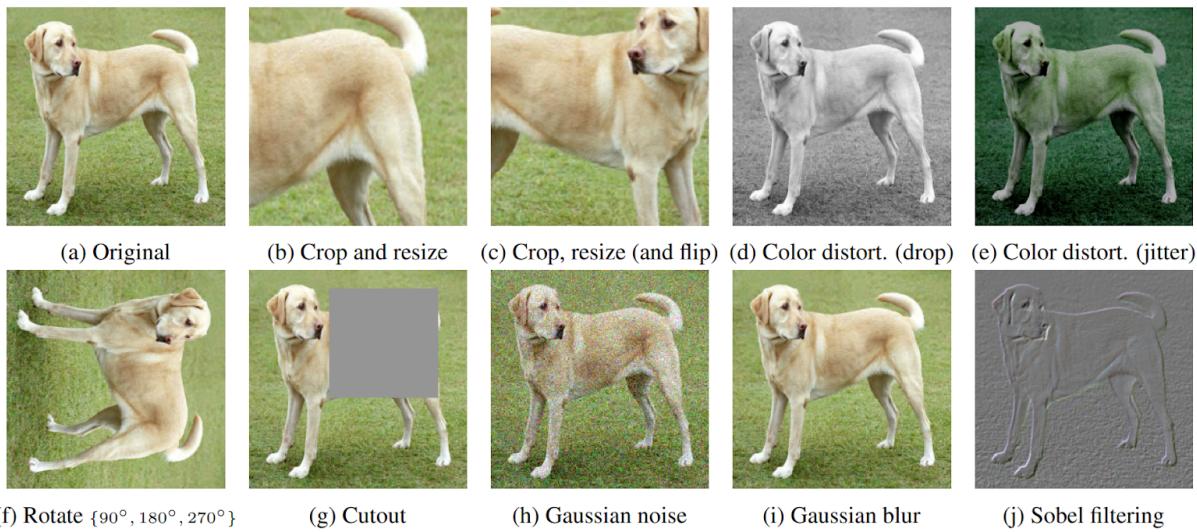
```
[3]: class ContrastiveTransformations(object):

    def __init__(self, base_transforms, n_views=2):
        self.base_transforms = base_transforms
        self.n_views = n_views

    def __call__(self, x):
        return [self.base_transforms(x) for i in range(self.n_views)]
```

The contrastive learning framework can easily be extended to have more *positive* examples by sampling more than two augmentations of the same image. However, the most efficient training is usually obtained by using only two.

Next, we can look at the specific augmentations we want to apply. The choice of the data augmentation to use is the most crucial hyperparameter in SimCLR since it directly affects how the latent space is structured, and what patterns might be learned from the data. Let's first take a look at some of the most popular data augmentations (figure credit - Ting Chen and Geoffrey Hinton):



All of them can be used, but it turns out that two augmentations stand out in their importance: crop-and-resize, and color distortion. Interestingly, however, they only lead to strong performance if they have been used together as discussed by Ting Chen et al. in their SimCLR paper. When performing randomly cropping and resizing, we can distinguish between two situations: (a) cropped image A provides a local view of cropped image B, or (b) cropped images C and D show neighboring views of the same image (figure credit - Ting Chen and Geoffrey Hinton).

While situation (a) requires the model to learn some sort of scale invariance to make crops A and B similar in latent space, situation (b) is more challenging since the model needs to recognize an object beyond its limited view. However, without color distortion, there is a loophole that the model can exploit, namely that different crops of the same image usually look very similar in color space. Consider the picture of the dog above. Simply from the color of the fur and the green color tone of the background, you can reason that two patches belong to the same image without actually recognizing the dog in the picture. In this case, the model might end up focusing only on the color histograms of the images, and ignore other more generalizable features. If, however, we distort the colors in the two patches randomly

and independently of each other, the model cannot rely on this simple feature anymore. Hence, by combining random cropping and color distortions, the model can only match two patches by learning generalizable representations.

Overall, for our experiments, we apply a set of 5 transformations following the original SimCLR setup: random horizontal flip, crop-and-resize, color distortion, random grayscale, and gaussian blur. In comparison to the [original implementation](#), we reduce the effect of the color jitter slightly (0.5 instead of 0.8 for brightness, contrast, and saturation, and 0.1 instead of 0.2 for hue). In our experiments, this setting obtained better performance and was faster and more stable to train. If, for instance, the brightness scale highly varies in a dataset, the original settings can be more beneficial since the model can't rely on this information anymore to distinguish between images.

```
[4]: contrast_transforms = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomResizedCrop(size=96),
    transforms.RandomApply([
        transforms.ColorJitter(brightness=0.5,
                              contrast=0.5,
                              saturation=0.5,
                              hue=0.1)
    ], p=0.8),
    transforms.RandomGrayscale(p=0.2),
    transforms.GaussianBlur(kernel_size=9),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

After discussing the data augmentation techniques, we can now focus on the dataset. In this tutorial, we will use the [STL10 dataset](#), which, similarly to CIFAR10, contains images of 10 classes: airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck. However, the images have a higher resolution, namely 96×96 pixels, and we are only provided with 500 labeled images per class. Additionally, we have a much larger set of 100,000 unlabeled images which are similar to the training images but are sampled from a wider range of animals and vehicles. This makes the dataset ideal to showcase the benefits that self-supervised learning offers.

Luckily, the STL10 dataset is provided through torchvision. Keep in mind, however, that since this dataset is relatively large and has a considerably higher resolution than CIFAR10, it requires more disk space (~3GB) and takes a bit of time to download. For our initial discussion of self-supervised learning and SimCLR, we will create two data loaders with our contrastive transformations above: the `unlabeled_data` will be used to train our model via contrastive learning, and `train_data_contrast` will be used as a validation set in contrastive learning.

```
[5]: unlabeled_data = STL10(root=DATASET_PATH, split='unlabeled', download=True,
                         transform=ContrastiveTransformations(contrast_transforms, n_
                         ↵views=2))
train_data_contrast = STL10(root=DATASET_PATH, split='train', download=True,
                           transform=ContrastiveTransformations(contrast_transforms, n_
                           ↵views=2))

Files already downloaded and verified
Files already downloaded and verified
```

Finally, before starting with our implementation of SimCLR, let's look at some example image pairs sampled with our augmentations:

```
[6]: # Visualize some examples
pl.seed_everything(42)
NUM_IMAGES = 6
imgs = torch.stack([img for idx in range(NUM_IMAGES) for img in unlabeled_data[idx][0]], ↵
                   dim=0)
```

(continues on next page)

(continued from previous page)

```
img_grid = torchvision.utils.make_grid(imgs, nrow=6, normalize=True, pad_value=0.9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(10,5))
plt.title('Augmented image examples of the STL10 dataset')
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

Global seed set to 42

Augmented image examples of the STL10 dataset



We see the wide variety of our data augmentation, including randomly cropping, grayscaling, gaussian blur, and color distortion. Thus, it remains a challenging task for the model to match two, independently augmented patches of the same image.

SimCLR implementation

Using the data loader pipeline above, we can now implement SimCLR. At each iteration, we get for every image x two differently augmented versions, which we refer to as \tilde{x}_i and \tilde{x}_j . Both of these images are encoded into a one-dimensional feature vector, between which we want to maximize similarity which minimizes it to all other images in the batch. The encoder network is split into two parts: a base encoder network $f(\cdot)$, and a projection head $g(\cdot)$. The base network is usually a deep CNN as we have seen in e.g. Tutorial 5 before, and is responsible for extracting a representation vector from the augmented data examples. In our experiments, we will use the common ResNet-18 architecture as $f(\cdot)$, and refer to the output as $f(\tilde{x}_i) = h_i$. The projection head $g(\cdot)$ maps the representation h into a space where we apply the contrastive loss, i.e., compare similarities between vectors. It is often chosen to be a small MLP with non-linearities, and for simplicity, we follow the original SimCLR paper setup by defining it as a two-layer MLP with ReLU activation in the hidden layer. Note that in the follow-up paper, SimCLRv2, the authors mention that larger/wider MLPs can boost the performance considerably. This is why we apply an MLP with four times larger hidden dimensions, but deeper MLPs showed to overfit on the given dataset. The general setup is visualized below (figure credit - Ting Chen et al.):

After finishing the training with contrastive learning, we will remove the projection head $g(\cdot)$, and use $f(\cdot)$ as a pre-trained feature extractor. The representations z that come out of the projection head $g(\cdot)$ have been shown to perform worse than those of the base network $f(\cdot)$ when finetuning the network for a new task. This is likely because the representations z are trained to become invariant to many features like the color that can be important for downstream tasks.

Thus, $g(\cdot)$ is only needed for the contrastive learning stage.

Now that the architecture is described, let's take a closer look at how we train the model. As mentioned before, we want to maximize the similarity between the representations of the two augmented versions of the same image, i.e., z_i and z_j in the figure above, while minimizing it to all other examples in the batch. SimCLR thereby applies the InfoNCE loss, originally proposed by Aaron van den Oord et al. for contrastive learning. In short, the InfoNCE loss compares the similarity of z_i and z_j to the similarity of z_i to any other representation in the batch by performing a softmax over the similarity values. The loss can be formally written as:

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(z_i, z_k)/\tau)} = -\text{sim}(z_i, z_j)/\tau + \log \left[\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(z_i, z_k)/\tau) \right]$$

The function sim is a similarity metric, and the hyperparameter τ is called temperature determining how peaked the distribution is. Since many similarity metrics are bounded, the temperature parameter allows us to balance the influence of many dissimilar image patches versus one similar patch. The similarity metric that is used in SimCLR is cosine similarity, as defined below:

$$\text{sim}(z_i, z_j) = \frac{z_i^\top \cdot z_j}{\|z_i\| \cdot \|z_j\|}$$

The maximum cosine similarity possible is 1, while the minimum is -1 . In general, we will see that the features of two different images will converge to a cosine similarity around zero since the minimum, -1 , would require z_i and z_j to be in the exact opposite direction in all feature dimensions, which does not allow for great flexibility.

Finally, now that we have discussed all details, let's implement SimCLR below as a PyTorch Lightning module:

```
[7]: class SimCLR(pl.LightningModule):

    def __init__(self, hidden_dim, lr, temperature, weight_decay, max_epochs=500):
        super().__init__()
        self.save_hyperparameters()
        assert self.hparams.temperature > 0.0, 'The temperature must be a positive float!'

        # Base model f(.)
        self.convnet = torchvision.models.resnet18(num_classes=4*hidden_dim) # Output of last linear layer
        # The MLP for g(.) consists of Linear->ReLU->Linear
        self.convnet.fc = nn.Sequential(
            self.convnet.fc, # Linear(ResNet output, 4*hidden_dim)
            nn.ReLU(inplace=True),
            nn.Linear(4*hidden_dim, hidden_dim)
        )

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(),
                               lr=self.hparams.lr,
                               weight_decay=self.hparams.weight_decay)
        lr_scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer,
                                                          T_max=self.hparams.max_epochs,
                                                          eta_min=self.hparams.lr/50)
        return [optimizer], [lr_scheduler]

    def info_nce_loss(self, batch, mode='train'):
        imgs, _ = batch
```

(continues on next page)

(continued from previous page)

```



```

Alternatively to performing the validation on the contrastive learning loss as well, we could also take a simple, small downstream task, and track the performance of the base network $f(\cdot)$ on that. However, in this tutorial, we will restrict ourselves to the STL10 dataset where we use the task of image classification on STL10 as our test task.

Training

Now that we have implemented SimCLR and the data loading pipeline, we are ready to train the model. We will use the same training function setup as usual. For saving the best model checkpoint, we track the metric `val_acc_top5`, which describes how often the correct image patch is within the top-5 most similar examples in the batch. This is usually less noisy than the top-1 metric, making it a better metric to choose the best model from.

```
[8]: def train_simclr(batch_size, max_epochs=500, **kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, 'SimCLR'),
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",

```

(continues on next page)

(continued from previous page)

```

        devices=1,
        max_epochs=max_epochs,
        callbacks=[ModelCheckpoint(save_weights_only=True, mode='max',
        monitor='val_acc_top5'),
                    LearningRateMonitor('epoch')])
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't
    need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, 'SimCLR.ckpt')
    if os.path.isfile(pretrained_filename):
        print(f'Found pretrained model at {pretrained_filename}, loading...')
        model = SimCLR.load_from_checkpoint(pretrained_filename) # Automatically loads
    ↪the model with the saved hyperparameters
    else:
        train_loader = data.DataLoader(unlabeled_data, batch_size=batch_size,
    ↪shuffle=True,
                                drop_last=True, pin_memory=True, num_workers=NUM_
    ↪WORKERS)
        val_loader = data.DataLoader(train_data_contrast, batch_size=batch_size,
    ↪shuffle=False,
                                drop_last=False, pin_memory=True, num_workers=NUM_
    ↪WORKERS)
        pl.seed_everything(42) # To be reproducible
        model = SimCLR(max_epochs=max_epochs, **kwargs)
        trainer.fit(model, train_loader, val_loader)
        model = SimCLR.load_from_checkpoint(trainer.checkpoint_callback.best_model_path)
    ↪# Load best checkpoint after training

    return model

```

A common observation in contrastive learning is that the larger the batch size, the better the models perform. A larger batch size allows us to compare each image to more negative examples, leading to overall smoother loss gradients. However, in our case, we experienced that a batch size of 256 was sufficient to get good results.

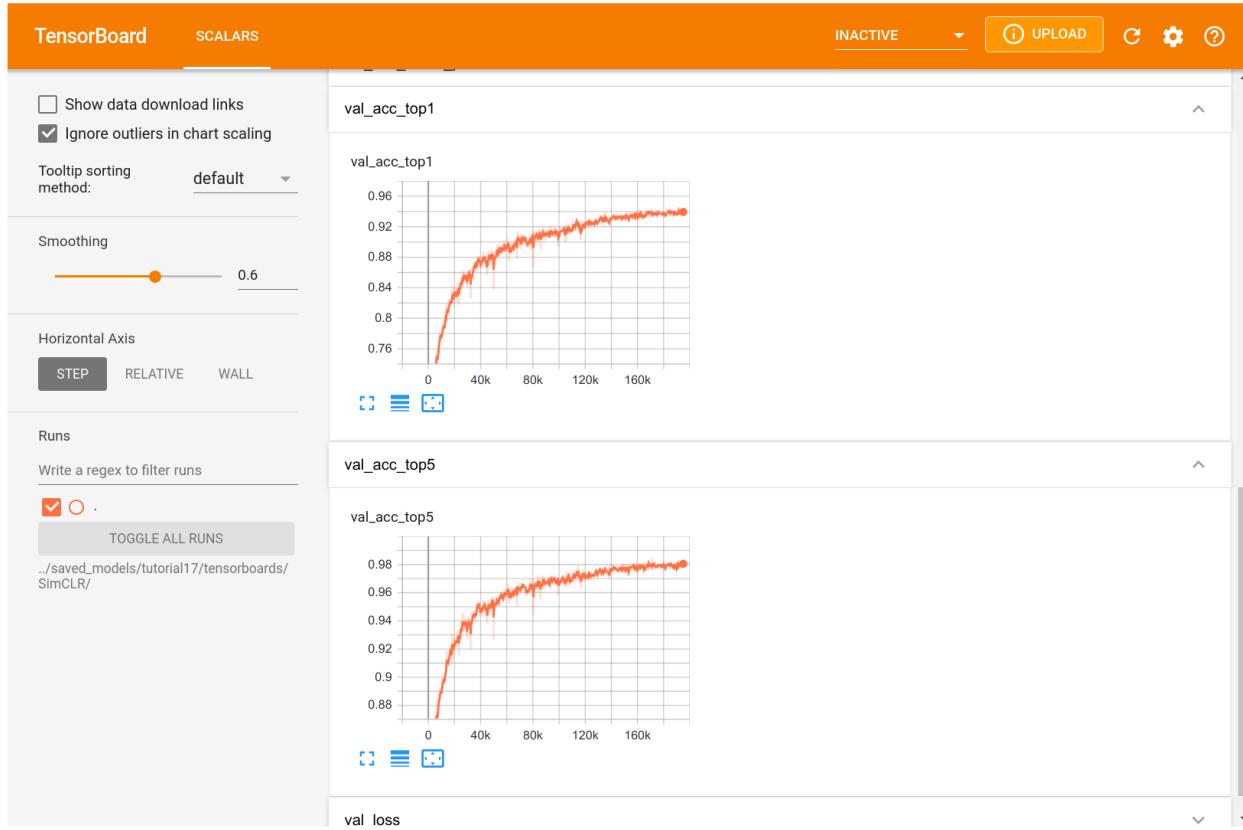
```
[9]: simclr_model = train_simclr(batch_size=256,
                               hidden_dim=128,
                               lr=5e-4,
                               temperature=0.07,
                               weight_decay=1e-4,
                               max_epochs=500)
```

GPU available: True, used: True
 TPU available: False, using: 0 TPU cores

Found pretrained model at ../saved_models/tutorial17/SimCLR.ckpt, loading...

To get an intuition of how training with contrastive learning behaves, we can take a look at the TensorBoard below:

```
[10]: %tensorboard --logdir ../saved_models/tutorial17/tensorboards/SimCLR/
```



One thing to note is that contrastive learning benefits a lot from long training. The shown plot above is from a training that took approx. 1 day on a NVIDIA TitanRTX. Training the model for even longer might reduce its loss further, but we did not experience any gains from it for the downstream task on image classification. In general, contrastive learning can also benefit from using larger models, if sufficient unlabeled data is available.

4.18.2 Logistic Regression

After we have trained our model via contrastive learning, we can deploy it on downstream tasks and see how well it performs with little data. A common setup, which also verifies whether the model has learned generalized representations, is to perform Logistic Regression on the features. In other words, we learn a single, linear layer that maps the representations to a class prediction. Since the base network $f(\cdot)$ is not changed during the training process, the model can only perform well if the representations of h describe all features that might be necessary for the task. Further, we do not have to worry too much about overfitting since we have very few parameters that are trained. Hence, we might expect that the model can perform well even with very little data.

First, let's implement a simple Logistic Regression setup for which we assume that the images already have been encoded in their feature vectors. If very little data is available, it might be beneficial to dynamically encode the images during training so that we can also apply data augmentations. However, the way we implement it here is much more efficient and can be trained within a few seconds. Further, using data augmentations did not show any significant gain in this simple setup.

```
[11]: class LogisticRegression(pl.LightningModule):
    def __init__(self, feature_dim, num_classes, lr, weight_decay, max_epochs=100):
        super().__init__()
        self.save_hyperparameters()
```

(continues on next page)

(continued from previous page)

```

# Mapping from representation h to classes
self.model = nn.Linear(feature_dim, num_classes)

def configure_optimizers(self):
    optimizer = optim.AdamW(self.parameters(),
                           lr=self.hparams.lr,
                           weight_decay=self.hparams.weight_decay)
    lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer,
                                                   milestones=[int(self.hparams.max_
→epochs*0.6),
                           int(self.hparams.max_
→epochs*0.8)],
                           gamma=0.1)
    return [optimizer], [lr_scheduler]

def _calculate_loss(self, batch, mode='train'):
    feats, labels = batch
    preds = self.model(feats)
    loss = F.cross_entropy(preds, labels)
    acc = (preds.argmax(dim=-1) == labels).float().mean()

    self.log(mode + '_loss', loss)
    self.log(mode + '_acc', acc)
    return loss

def training_step(self, batch, batch_idx):
    return self._calculate_loss(batch, mode='train')

def validation_step(self, batch, batch_idx):
    self._calculate_loss(batch, mode='val')

def test_step(self, batch, batch_idx):
    self._calculate_loss(batch, mode='test')

```

The data we use is the training and test set of STL10. The training contains 500 images per class, while the test set has 800 images per class.

```
[12]: img_transforms = transforms.Compose([transforms.ToTensor(),
                                         transforms.Normalize((0.5,), (0.5,))])

train_img_data = STL10(root=DATASET_PATH, split='train', download=True,
                      transform=img_transforms)
test_img_data = STL10(root=DATASET_PATH, split='test', download=True,
                      transform=img_transforms)

print("Number of training examples:", len(train_img_data))
print("Number of test examples:", len(test_img_data))

Files already downloaded and verified
Files already downloaded and verified
Number of training examples: 5000
Number of test examples: 8000
```

Next, we implement a small function to encode all images in our datasets. The output representations are then used as inputs to the Logistic Regression model.

```
[13]: @torch.no_grad()
def prepare_data_features(model, dataset):
    # Prepare model
    network = deepcopy(model.convnet)
    network.fc = nn.Identity()  # Removing projection head g(.)
    network.eval()
    network.to(device)

    # Encode all images
    data_loader = data.DataLoader(dataset, batch_size=64, num_workers=NUM_WORKERS,
    ↵shuffle=False, drop_last=False)
    feats, labels = [], []
    for batch_imgs, batch_labels in tqdm(data_loader):
        batch_imgs = batch_imgs.to(device)
        batch_feats = network(batch_imgs)
        feats.append(batch_feats.detach().cpu())
        labels.append(batch_labels)

    feats = torch.cat(feats, dim=0)
    labels = torch.cat(labels, dim=0)

    # Sort images by labels
    labels, idxs = labels.sort()
    feats = feats[idxs]

    return data.TensorDataset(feats, labels)
```

Let's apply the function to both training and test set below.

```
[14]: train_feats_simclr = prepare_data_features(simclr_model, train_img_data)
test_feats_simclr = prepare_data_features(simclr_model, test_img_data)

0% | 0/79 [00:00<?, ?it/s]
/home/phillip/anaconda3/envs/dl2020/lib/python3.7/site-packages/torch/nn/functional.py:
    ↵718: UserWarning: Named tensors and all their associated APIs are an experimental
    ↵feature and subject to change. Please do not use them for anything important until
    ↵they are released as stable. (Triggered internally at  /opt/conda/conda-bld/pytorch_
    ↵1623448265233/work/c10/core/TensorImpl.h:1156.)
    return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)

0% | 0/125 [00:00<?, ?it/s]
```

Finally, we can write a training function as usual. We evaluate the model on the test set every 10 epochs to allow early stopping, but the low frequency of the validation ensures that we do not overfit too much on the test set.

```
[15]: def train_logreg(batch_size, train_feats_data, test_feats_data, model_suffix, max_
    ↵epochs=100, **kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH,
    ↵"LogisticRegression"),
        accelerator="gpu" if str(device).startswith("cuda") else "cpu",
        devices=1,
```

(continues on next page)

(continued from previous page)

```

        max_epochs=max_epochs,
        callbacks=[ModelCheckpoint(save_weights_only=True, mode='max',  

        monitor='val_acc'),  

                    LearningRateMonitor("epoch")],  

        enable_progress_bar=False,  

        check_val_every_n_epoch=10)
    trainer.logger._default_hp_metric = None

    # Data loaders
    train_loader = data.DataLoader(train_feats_data, batch_size=batch_size, shuffle=True,  

                                    drop_last=False, pin_memory=True, num_workers=0)
    test_loader = data.DataLoader(test_feats_data, batch_size=batch_size, shuffle=False,  

                                drop_last=False, pin_memory=True, num_workers=0)

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, f"LogisticRegression_{model_"
    suffix}.ckpt")
    if os.path.isfile(pretrained_filename):
        print(f"Found pretrained model at {pretrained_filename}, loading...")
        model = LogisticRegression.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything(42) # To be reproducible
        model = LogisticRegression(**kwargs)
        trainer.fit(model, train_loader, test_loader)
        model = LogisticRegression.load_from_checkpoint(trainer.checkpoint_callback.best_
model_path)

    # Test best model on train and validation set
    train_result = trainer.test(model, train_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"train": train_result[0]["test_acc"], "test": test_result[0]["test_acc"]}

    return model, result

```

Despite the training dataset of STL10 already only having 500 labeled images per class, we will perform experiments with even smaller datasets. Specifically, we train a Logistic Regression model for datasets with only 10, 20, 50, 100, 200, and all 500 examples per class. This gives us an intuition on how well the representations learned by contrastive learning can be transferred to a image recognition task like this classification. First, let's define a function to create the intended sub-datasets from the full training set:

```
[16]: def get_smaller_dataset(original_dataset, num_imgs_per_label):
    new_dataset = data.TensorDataset(
        *[t.unflatten(0, (10, -1))[:, :num_imgs_per_label].flatten(0, 1) for t in_
        original_dataset.tensors]
    )
    return new_dataset
```

Next, let's run all models. Despite us training 6 models, this cell could be run within a minute or two without the pretrained models.

```
[17]: results = {}
for num_imgs_per_label in [10, 20, 50, 100, 200, 500]:
```

(continues on next page)

(continued from previous page)

```

sub_train_set = get_smaller_dataset(train_feats_simclr, num_imgs_per_label)
_, small_set_results = train_logreg(batch_size=64,
                                     train_feats_data=sub_train_set,
                                     test_feats_data=test_feats_simclr,
                                     model_suffix=num_imgs_per_label,
                                     feature_dim=train_feats_simclr.tensors[0].
                                     shape[1],
                                     num_classes=10,
                                     lr=1e-3,
                                     weight_decay=1e-3)
results[num_imgs_per_label] = small_set_results

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
/home/phillip/anaconda3/envs/dl2020/lib/python3.7/site-packages/pytorch_lightning/
↳ utilities/distributed.py:69: UserWarning: Your test_dataloader has `shuffle=True`, it
↳ is best practice to turn this off for val/test/predict dataloaders.
    warnings.warn(*args, **kwargs)
/home/phillip/anaconda3/envs/dl2020/lib/python3.7/site-packages/pytorch_lightning/
↳ utilities/distributed.py:69: UserWarning: The dataloader, test dataloader 0, does not
↳ have many workers which may be a bottleneck. Consider increasing the value of the `num_
↳ workers` argument` (try 16 which is the number of cpus on this machine) in the
↳ `DataLoader` init to improve performance.
    warnings.warn(*args, **kwargs)
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]

Found pretrained model at ../saved_models/tutorial17/LogisticRegression_10.ckpt, ↳
↳ loading...
Found pretrained model at ../saved_models/tutorial17/LogisticRegression_20.ckpt, ↳
↳ loading...

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]

Found pretrained model at ../saved_models/tutorial17/LogisticRegression_50.ckpt, ↳
↳ loading...
Found pretrained model at ../saved_models/tutorial17/LogisticRegression_100.ckpt, ↳
↳ loading...

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]

```

(continues on next page)

(continued from previous page)

```

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]

Found pretrained model at ../saved_models/tutorial17/LogisticRegression_200.ckpt, ↴
↳ loading...
Found pretrained model at ../saved_models/tutorial17/LogisticRegression_500.ckpt, ↴
↳ loading...

LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]

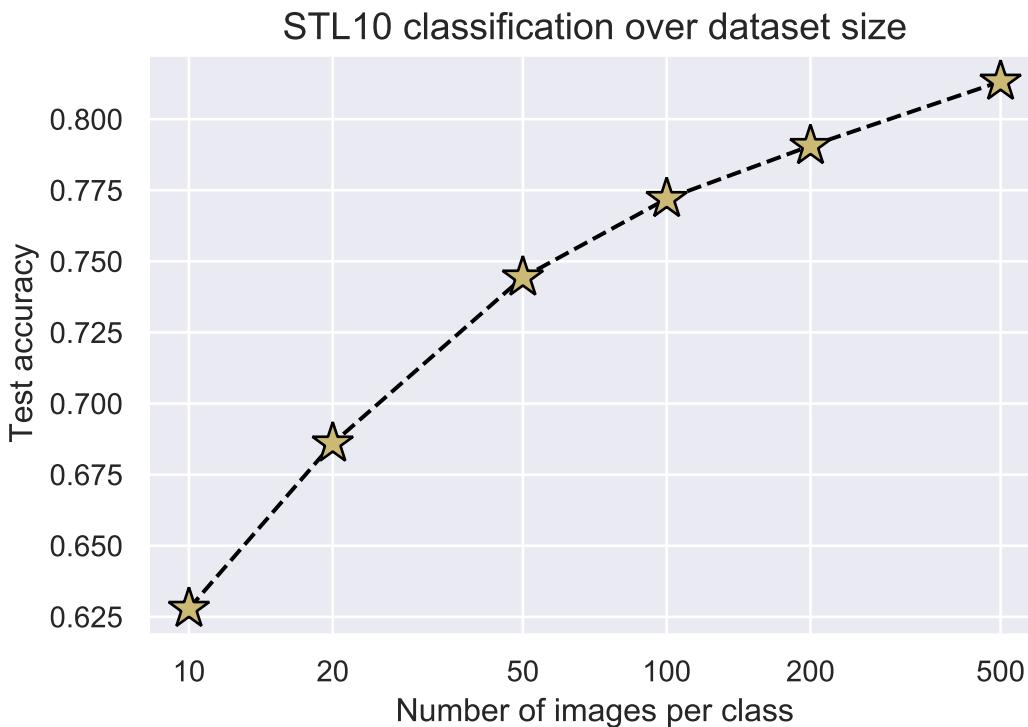
```

Finally, let's plot the results.

```
[18]: dataset_sizes = sorted([k for k in results])
test_scores = [results[k]["test"] for k in dataset_sizes]

fig = plt.figure(figsize=(6,4))
plt.plot(dataset_sizes, test_scores, '--', color="#000", marker="*", markeredgecolor="→#000", markerfacecolor="y", markersize=16)
plt.xscale("log")
plt.xticks(dataset_sizes, labels=dataset_sizes)
plt.title("STL10 classification over dataset size", fontsize=14)
plt.xlabel("Number of images per class")
plt.ylabel("Test accuracy")
plt.minorticks_off()
plt.show()

for k, score in zip(dataset_sizes, test_scores):
    print(f'Test accuracy for {k:3d} images per label: {100*score:.2f}%')
```



```
Test accuracy for 10 images per label: 62.79%
Test accuracy for 20 images per label: 68.60%
Test accuracy for 50 images per label: 74.44%
Test accuracy for 100 images per label: 77.20%
Test accuracy for 200 images per label: 79.06%
Test accuracy for 500 images per label: 81.33%
```

As one would expect, the classification performance improves the more data we have. However, with only 10 images per class, we can already classify more than 60% of the images correctly. This is quite impressive, considering that the images are also higher dimensional than e.g. CIFAR10. With the full dataset, we achieve an accuracy of 81%. The increase between 50 to 500 images per class might suggest a linear increase in performance with an exponentially larger dataset. However, with even more data, we could also finetune $f(\cdot)$ in the training process, allowing for the representations to adapt more to the specific classification task given.

To set the results above into perspective, we will train the base network, a ResNet-18, on the classification task from scratch.

4.18.3 Baseline

As a baseline to our results above, we will train a standard ResNet-18 with random initialization on the labeled training set of STL10. The results will give us an indication of the advantages that contrastive learning on unlabeled data has compared to using only supervised training. The implementation of the model is straightforward since the ResNet architecture is provided in the torchvision library.

```
[19]: class ResNet(pl.LightningModule):

    def __init__(self, num_classes, lr, weight_decay, max_epochs=100):
        super().__init__()
        self.save_hyperparameters()
        self.model = torchvision.models.resnet18(num_classes=num_classes)

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(),
                               lr=self.hparams.lr,
                               weight_decay=self.hparams.weight_decay)
        lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer,
                                                       milestones=[int(self.hparams.max_
→epochs*0.7),
                           int(self.hparams.max_
→epochs*0.9)],
                           gamma=0.1)
        return [optimizer], [lr_scheduler]

    def _calculate_loss(self, batch, mode='train'):
        imgs, labels = batch
        preds = self.model(imgs)
        loss = F.cross_entropy(preds, labels)
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        self.log(mode + '_loss', loss)
        self.log(mode + '_acc', acc)
        return loss
```

(continues on next page)

(continued from previous page)

```

def training_step(self, batch, batch_idx):
    return self._calculate_loss(batch, mode='train')

def validation_step(self, batch, batch_idx):
    self._calculate_loss(batch, mode='val')

def test_step(self, batch, batch_idx):
    self._calculate_loss(batch, mode='test')

```

It is clear that the ResNet easily overfits on the training data since its parameter count is more than 1000 times larger than the dataset size. To make the comparison to the contrastive learning models fair, we apply data augmentations similar to the ones we used before: horizontal flip, crop-and-resize, grayscale, and gaussian blur. Color distortions as before are not used because the color distribution of an image showed to be an important feature for the classification. Hence, we observed no noticeable performance gains when adding color distortions to the set of augmentations. Similarly, we restrict the resizing operation before cropping to the max. 125% of its original resolution, instead of 1250% as done in SimCLR. This is because, for classification, the model needs to recognize the full object, while in contrastive learning, we only want to check whether two patches belong to the same image/object. Hence, the chosen augmentations below are overall weaker than in the contrastive learning case.

```
[20]: train_transforms = transforms.Compose([transforms.RandomHorizontalFlip(),
                                         transforms.RandomResizedCrop(size=96, scale=(0.8, 1.0)),
                                         transforms.RandomGrayscale(p=0.2),
                                         transforms.GaussianBlur(kernel_size=9, sigma=(0.1, 0.5)),
                                         transforms.ToTensor(),
                                         transforms.Normalize((0.5,), (0.5,))])

train_img_aug_data = STL10(root=DATASET_PATH, split='train', download=True,
                           transform=train_transforms)

Files already downloaded and verified
```

The training function for the ResNet is almost identical to the Logistic Regression setup. Note that we allow the ResNet to perform validation every 2 epochs to also check whether the model overfits strongly in the first iterations or not.

```
[21]: def train_resnet(batch_size, max_epochs=100, **kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "ResNet"),
                          accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                          devices=1,
                          max_epochs=max_epochs,
                          callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
                                                      monitor="val_acc"),
                                     LearningRateMonitor("epoch")],
                          check_val_every_n_epoch=2)
    trainer.logger._default_hp_metric = None

    # Data loaders
    train_loader = data.DataLoader(train_img_aug_data, batch_size=batch_size,
                                   shuffle=True,
                                   drop_last=True, pin_memory=True, num_workers=NUM_WORKERS)
```

(continues on next page)

(continued from previous page)

```

test_loader = data.DataLoader(test_img_data, batch_size=batch_size, shuffle=False,
                             drop_last=False, pin_memory=True, num_workers=NUM_
                             ↪WORKERS)

# Check whether pretrained model exists. If yes, load it and skip training
pretrained_filename = os.path.join(CHECKPOINT_PATH, "ResNet.ckpt")
if os.path.isfile(pretrained_filename):
    print("Found pretrained model at %s, loading..." % pretrained_filename)
    model = ResNet.load_from_checkpoint(pretrained_filename)
else:
    pl.seed_everything(42) # To be reproducible
    model = ResNet(**kwargs)
    trainer.fit(model, train_loader, test_loader)
    model = ResNet.load_from_checkpoint(trainer.callback.best_model_path)

# Test best model on validation set
train_result = trainer.test(model, train_loader, verbose=False)
val_result = trainer.test(model, test_loader, verbose=False)
result = {"train": train_result[0]["test_acc"], "test": val_result[0]["test_acc"]}

return model, result

```

Finally, let's train the model and check its results:

```

[22]: resnet_model, resnet_result = train_resnet(batch_size=64,
                                                num_classes=10,
                                                lr=1e-3,
                                                weight_decay=2e-4,
                                                max_epochs=100)
print(f"Accuracy on training set: {100*resnet_result['train']:.2f}%")
print(f"Accuracy on test set: {100*resnet_result['test']:.2f}%")

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]

Found pretrained model at ../saved_models/tutorial17/ResNet.ckpt, loading...

/home/phillip/anaconda3/envs/dl2020/lib/python3.7/site-packages/pytorch_lightning/
↪utilities/distributed.py:69: UserWarning: Your test_dataloader has `shuffle=True`, it ↪
↪is best practice to turn this off for val/test/predict dataloaders.
    warnings.warn(*args, **kwargs)

Testing: 0it [00:00, ?it/s]
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]

Testing: 0it [00:00, ?it/s]

Accuracy on training set: 99.76%
Accuracy on test set: 73.31%

```

The ResNet trained from scratch achieves 73.31% on the test set. This is almost 8% less than the contrastive learning model, and even slightly less than SimCLR achieves with 1/10 of the data. This shows that self-supervised, contrastive learning provides considerable performance gains by leveraging large amounts of unlabeled data when little labeled data is available.

4.18.4 Conclusion

In this tutorial, we have discussed self-supervised contrastive learning and implemented SimCLR as an example method. We have applied it to the STL10 dataset and showed that it can learn generalizable representations that we can use to train simple classification models. With 500 images per label, it achieved an 8% higher accuracy than a similar model solely trained from supervision and performs on par with it when only using a tenth of the labeled data. Our experimental results are limited to a single dataset, but recent works such as [Ting Chen et al.](#) showed similar trends for larger datasets like ImageNet. Besides the discussed hyperparameters, the size of the model seems to be important in contrastive learning as well. If a lot of unlabeled data is available, larger models can achieve much stronger results and come close to their supervised baselines. Further, there are also approaches for combining contrastive and supervised learning, leading to performance gains beyond supervision (see [Khosla et al.](#)). Moreover, contrastive learning is not the only approach to self-supervised learning that has come up in the last two years and showed great results. Other methods include distillation-based methods like [BYOL](#) and redundancy reduction techniques like [Barlow Twins](#). There is a lot more to explore in the self-supervised domain, and more, impressive steps ahead are to be expected.

References

- [1] Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. (2020). A simple framework for contrastive learning of visual representations. In International Conference on Machine Learning (ICML 2020). PMLR. ([link](#))
- [2] Chen, T., Kornblith, S., Swersky, K., Norouzi, M., and Hinton, G. (2020). Big self-supervised models are strong semi-supervised learners. NeurIPS 2021 ([link](#)).
- [3] Oord, A. V. D., Li, Y., and Vinyals, O. (2018). Representation learning with contrastive predictive coding. arXiv preprint arXiv:1807.03748. ([link](#))
- [4] Grill, J.B., Strub, F., Altché, F., Tallec, C., Richemond, P.H., Buchatskaya, E., Doersch, C., Pires, B.A., Guo, Z.D., Azar, M.G. and Piot, B. (2020). Bootstrap your own latent: A new approach to self-supervised learning. NeurIPS 2020 ([link](#))
- [5] Khosla, P., Teterwak, P., Wang, C., Sarna, A., Tian, Y., Isola, P., Maschinot, A., Liu, C. and Krishnan, D. (2020). Supervised contrastive learning. NeurIPS 2020 ([link](#))
- [6] Zbontar, J., Jing, L., Misra, I., LeCun, Y. and Deny, S. (2021). Barlow twins: Self-supervised learning via redundancy reduction. In International Conference on Machine Learning (ICML 2021). ([link](#))

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.19 Tutorial 2 (JAX): Introduction to JAX+Flax

Filled notebook:

Author: Phillip Lippe

Welcome to our JAX tutorial for the Deep Learning course at the University of Amsterdam! The following notebook is meant to give a short introduction to JAX, including writing and training your own neural networks with [Flax](#). But why should you learn JAX, if there are already so many other deep learning frameworks like [PyTorch](#) and [TensorFlow](#)? The short answer: because it can be extremely fast. For instance, a small GoogleNet on CIFAR10, which we discuss in detail in [Tutorial 5](#), can be trained in JAX 3x faster than in PyTorch with a similar setup. Note that for larger models, larger batch sizes, or smaller GPUs, a considerably smaller speedup is expected, and the code has not been designed for benchmarking. Nonetheless, JAX enables this speedup by compiling functions and numerical programs for accelerators (GPU/TPU) *just in time*, finding the optimal utilization of the hardware. Frameworks with dynamic computation graphs like PyTorch cannot achieve the same efficiency, since they cannot anticipate the next operations before the user calls them. For example, in an Inception block of GoogleNet, we apply multiple convolutional layers in parallel on the same input. JAX can optimize the execution of this layer by compiling the whole forward pass for the available accelerator and fusing operations where possible, reducing memory access and speeding up execution. In contrast, when calling the first convolutional layer in PyTorch, the framework does not know that multiple convolutions on the same feature map will follow. It sends each operation one by one to the GPU, and can only adapt the execution after seeing the next Python calls. Hence, JAX can make more efficient use of the GPU than, for instance, PyTorch.

However, everything comes with a price. In order to efficiently compile programs just-in-time in JAX, the functions need to be written with certain constraints. Firstly, the functions are not allowed to have side-effects, meaning that they are not allowed to affect any variable outside of their namespaces. For instance, in-place operations affect a variable even outside of the function. Moreover, stochastic operations such as `torch.rand(...)` change the global state of pseudo random number generators, which is not allowed in functional JAX (we will see later how JAX handles random number generation). Secondly, JAX compiles the functions based on anticipated shapes of all arrays/tensors in the function. This becomes problematic if the shapes or the program flow within the function depends on the values of the tensor. For instance, in the operation `y = x[x>3]`, the shape of `y` depends on how many values of `x` are greater than 3. We will discuss more of these constraints in this notebook. Still, in most common cases of training neural networks, it is straightforward to write functions within these constraints.

This tutorial is heavily inspired by many great JAX tutorials before, and a (non-exclusive) list of them are:

- [JAX 101](#) with many subtutorials on individual parts of JAX
- [JAX - The Sharp Bits](#) discusses the constraints of JAX and how to overcome them
- [Jax for the Impatient](#) for a quick intro to JAX with focus on deep learning
- [Flax Basics](#) as introduction to the Flax framework

Throughout this tutorial, we will draw comparisons to PyTorch and also use its data loading library (see our [PyTorch tutorial](#) for a refresher). JAX is not meant to ‘redefine the wheel’, so we can combine it framework-agnostic parts from PyTorch (e.g., data loading) and TensorFlow (e.g., logging in TensorBoard). Further, we use [Flax](#) as a neural network library in JAX, and [Optax](#) to implement common deep learning optimizers. More on them later in the notebook. First, let’s get started with some basic JAX operations.

```
[1]: ## Standard libraries
import os
import math
import numpy as np
import time

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgba
import seaborn as sns
sns.set()
```

(continues on next page)

(continued from previous page)

```
## Progress bar
from tqdm.auto import tqdm
```

4.19.1 JAX as NumPy on accelerators

Every deep learning framework has its own API for dealing with data arrays. For example, PyTorch uses `torch.Tensor` as data arrays on which it defines several operations like matrix multiplication, taking the mean of the elements, etc. In JAX, this basic API strongly resembles the one of `NumPy`, and even has the same name in JAX (`jax.numpy`). So, for now, let's think of JAX as NumPy that runs on accelerators. As a first step, let's import JAX and its NumPy API:

```
[2]: import jax
import jax.numpy as jnp
print("Using jax", jax.__version__)

Using jax 0.3.13
```

At the current time of writing (June 2022), the newest JAX version is `0.3.13` which supports most of the common NumPy functionalities. The NumPy API of JAX is usually imported as `jnp`, to keep a resemblance to NumPy's import as `np`. In the following subsections, we will discuss the main differences between the classical NumPy API and the one of JAX.

Device Arrays

As a first test, let's create some arbitrary arrays like we would do in NumPy. For instance, let's create an array of zeros with shape `[2, 5]`:

```
[3]: a = jnp.zeros((2, 5), dtype=jnp.float32)
print(a)

[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Similarly, we can create an array with values of 0 to 5 by using `arange`:

```
[4]: b = jnp.arange(6)
print(b)

[0 1 2 3 4 5]
```

You might now wonder whether the arrays `a` and `b` are simply NumPy arrays. To check that, let's print out the class of `b`:

```
[5]: b.__class__
[5]: jaxlib.xla_extension.DeviceArray
```

Instead of a simple NumPy array, it shows the type `DeviceArray` which is what JAX uses to represent arrays. In contrast to NumPy, JAX can execute the same code on different backends – CPU, GPU and TPU. A `DeviceArray` therefore represents an array which is on one of the backends. Similar to PyTorch, we can check the device of an array by calling `.device()`:

```
[6]: b.device()
```

```
[6]: GpuDevice(id=0, process_index=0)
```

As you can see, the array `b` is already natively on a GPU although we did not specify this explicitly as you would do in PyTorch (on Colab, remember to select a GPU in your runtime environment). In order to change the device of an array, we can use `jax.device_get`:

```
[7]: b_cpu = jax.device_get(b)
```

```
print(b_cpu.__class__)
```

```
<class 'numpy.ndarray'>
```

Unsurprisingly, a simple CPU-based array is nothing else than a NumPy array, which allows for a simple conversion between the two frameworks! To explicitly push a NumPy array to the accelerator, you can use `jax.device_put`:

```
[8]: b_gpu = jax.device_put(b_cpu)
print(f'Device put: {b_gpu.__class__} on {b_gpu.device()}' )
```

```
Device put: <class 'jaxlib.xla_extension.DeviceArray'> on gpu:0
```

Nicely enough, JAX will handle any device clash itself when you try to perform operations on a NumPy array and a `DeviceArray` by modeling the output as `DeviceArray` again:

```
[9]: b_cpu + b_gpu
```

```
[9]: DeviceArray([ 0,  2,  4,  6,  8, 10], dtype=int32)
```

Finally, we can also print all our available devices using `jax.devices`:

```
[10]: jax.devices()
```

```
[10]: [GpuDevice(id=0, process_index=0), GpuDevice(id=1, process_index=0)]
```

A technical detail of running operations on `DeviceArrays` is that when a JAX function is called, the corresponding operation takes place asynchronously on the accelerator when possible. For instance, if we call `out = jnp.matmul(b, b)`, JAX first returns a placeholder array for `out` which may not be filled with the values as soon as the function calls finishes. This way, Python will not block the execution of follow-up statements, but instead only does it whenever we strictly need the value of `out`, for instance for printing or putting it on CPU. PyTorch uses a very similar principle to allow asynchronous computation. For more details, see [JAX - Asynchronous Dispatch](#).

Immutable tensors

When we would like to change a NumPy array in-place, like replacing the first element of `b` with 1 instead of 0, we could simply write `b[0]=1`. However, in JAX, this is not possible. A `DeviceArray` object is *immutable*, which means that no in-place operations are possible. The reason for this goes back to our discussion in the introduction: JAX requires programs to be “pure” functions, i.e. no effects on variables outside of the function are allowed. Allowing in-place operations of variables would make the program analysis for JAX’s just-in-time compilation difficult. Instead, we can use the expression `b.at[0].set(1)` which, analogous to the in-place operation, returns a new array which is identical to `b`, except that its value at the first position is 1. Let’s try that out below:

```
[11]: b_new = b.at[0].set(1)
print('Original array:', b)
print('Changed array:', b_new)
```

```
Original array: [0 1 2 3 4 5]
Changed array: [1 1 2 3 4 5]
```

However, we said that JAX is very efficient. Isn't creating a new array in this case the opposite? While it is indeed less efficient, it can be made much more efficient with JAX's just-in-time compilation. The compiler can recognize unnecessary array duplications, and replace them with in-place operations again. More on the just-in-time compilation later!

Pseudo Random Numbers in JAX

In machine learning, we come across several situations where we need to generate pseudo random numbers: randomly shuffling a dataset, sampling a dropout mask for regularization, training a VAE by sampling from the approximate posterior, etc. In libraries like NumPy and PyTorch, the random number generator are controlled by a seed, which we set initially to obtain the same samples every time we run the code (this is why the numbers are not truly random, hence "pseudo"-random). However, if we call `np.random.normal()` 5 times consecutively, we will get 5 different numbers since every execution changes the state/seed of the pseudo random number generation (PRNG). In JAX, if we would try to generate a random number with this approach, a function creating pseudo-random number would have an effect outside of it. To prevent this, JAX takes a different approach by explicitly passing and iterating the PRNG state. First, let's create a PRNG for the seed 42:

```
[12]: rng = jax.random.PRNGKey(42)
```

Now, we can use this PRNG state to generate random numbers. Since with this state, the random number generation becomes deterministic, we sample the same number every time. This is not the case in NumPy if we set the seed once before both operations:

```
[13]: # A non-desirable way of generating pseudo-random numbers...
jax_random_number_1 = jax.random.normal(rng)
jax_random_number_2 = jax.random.normal(rng)
print('JAX - Random number 1:', jax_random_number_1)
print('JAX - Random number 2:', jax_random_number_2)

# Typical random numbers in NumPy
np.random.seed(42)
np_random_number_1 = np.random.normal()
np_random_number_2 = np.random.normal()
print('NumPy - Random number 1:', np_random_number_1)
print('NumPy - Random number 2:', np_random_number_2)

JAX - Random number 1: -0.18471177
JAX - Random number 2: -0.18471177
NumPy - Random number 1: 0.4967141530112327
NumPy - Random number 2: -0.13826430117118466
```

Usually, we want to have a behavior like NumPy where we get a different random number every time we sample. To achieve this, we can *split* the PRNG state to get usable subkeys every time we need a new pseudo-random number. We can do this with `jax.random.split(...)`:

```
[14]: rng, subkey1, subkey2 = jax.random.split(rng, num=3) # We create 3 new keys
jax_random_number_1 = jax.random.normal(subkey1)
jax_random_number_2 = jax.random.normal(subkey2)
print('JAX new - Random number 1:', jax_random_number_1)
print('JAX new - Random number 2:', jax_random_number_2)
```

```
JAX new - Random number 1: 0.107961535
JAX new - Random number 2: -1.2226542
```

Every time you run this cell, you will obtain different random numbers for both operations since we create new PRNG states before sampling. In general, you want to split the PRNG key every time before generating a pseudo-number, to prevent accidentally obtaining the exact same numbers (for instance, sampling the exact same dropout mask every time you run the network makes dropout itself quite useless...). For a deeper dive into the ideas behind the random number generation in JAX, see JAX's tutorial on [Pseudo Random Numbers](#).

4.19.2 Function transformations with Jaxpr

Rosalia Schneider and Vladimir Mikulik summarize the key points of JAX in the [JAX 101](#) tutorial as follows:

The most important difference, and in some sense the root of all the rest, is that JAX is designed to be functional, as in functional programming. The reason behind this is that the kinds of program transformations that JAX enables are much more feasible in functional-style programs. [...] The important feature of functional programming to grok when working with JAX is very simple: don't write code with side-effects.

Essentially, we want to write our main code of JAX in functions that do not affect anything else besides its outputs. For instance, we do not want to change input arrays in-place, or access global variables. While this might seem limiting at first, you get used to this quite quickly and most JAX functions that need to fulfill these constraints can be written this way without problems. Note that not all possible functions in training a neural network need to fulfill the constraints. For instance, loading or saving of models, the logging, or the data generation can be done in naive functions. Only the network execution, which we want to do very efficiently on our accelerator (GPU or TPU), should strictly follow these constraints.

What does make JAX functions so special, and how can we think about them? A good way of gaining understanding in how JAX handles function is to understand its intermediate representation: jaxpr. Conceptually, you can think of any operation that JAX does on a function, as first trace-specializing the Python function to be transformed into a small and well-behaved intermediate form. This means that we check which operations are performed on which array, and what shapes the arrays are. Based on this representation, JAX then interprets the function with transformation-specific interpretation rules, which includes automatic differentiation or compiling a function in XLA to efficiently use the accelerator.

To illustrate this intermediate representation, let's consider the same simple function we used in the [PyTorch](#) tutorial to discuss the concept of dynamic computation graphs:

$$y = \frac{1}{|x|} \sum_i [(x_i + 2)^2 + 3]$$

Using common NumPy operations in JAX, we can write it as follows:

```
[15]: def simple_graph(x):
    x = x + 2
    x = x ** 2
    x = x + 3
    y = x.mean()
    return y

inp = jnp.arange(3, dtype=jnp.float32)
print('Input', inp)
print('Output', simple_graph(inp))

Input [0. 1. 2.]
Output 12.666667
```

To view the jaxpr representation of this function, we can use `jax.make_jaxpr`. Since the tracing depends on the shape of the input, we need to pass an input to the function (here of shape [3]):

```
[16]: jax.make_jaxpr(simple_graph)(inp)
[16]: { lambda ; a:f32[3]. let
         b:f32[3] = add a 2.0
         c:f32[3] = integer_pow[y=2] b
         d:f32[3] = add c 3.0
         e:f32[] = reduce_sum[axes=(0,)] d
         f:f32[] = div e 3.0
     in (f,) }
```

A jaxpr representation follows the structure:

```
jaxpr ::= { lambda Var* ; Var+.
            let Eqn*
            in [Expr+] }
```

where `Var*` are constants and `Var+` are input arguments. In the cell above, this is `a:f32[3]`, i.e. an array of shape 3 with type `jnp.float32` (`inp`). The list of equations, `Eqn*`, define the intermediate results of the function. You can see that each operation in `simple_graph` is translated to a corresponding equation, like `x = x + 2` is translated to `b:f32[3] = add a 2.0`. Furthermore, you see the specialization of the operations on the input shape, like `x.mean()` being replacing in `e` and `f` with summing and dividing by 3. Finally, `Expr+` in the jaxpr representation are the outputs of the functions. In the example, this is `f`, i.e. the final result of the function. Based on these atomic operations, JAX offers all kind of function transformations, of which we will discuss the most important ones later in this section. Hence, you can consider the jaxpr representation is an intermediate compilation stage of JAX. What happens if we actually try to look at the jaxpr representation of a function with side-effect? Let's consider the following function, which, as an illustrative example, appends the input to a global list:

```
[17]: global_list = []

# Invalid function with side-effect
def norm(x):
    global_list.append(x)
    x = x ** 2
    n = x.sum()
    n = jnp.sqrt(n)
    return n

jax.make_jaxpr(norm)(inp)
```

```
[17]: { lambda ; a:f32[3]. let
         b:f32[3] = integer_pow[y=2] a
         c:f32[] = reduce_sum[axes=(0,)] b
         d:f32[] = sqrt c
     in (d,) }
```

As you can see, the jaxpr representation of the function does not contain any operation for `global_list.append(x)`. This is because jaxpr only understand side-effect-free code, and cannot represent such effects. Thus, we need to stick with pure functions without any side effects, to prevent any unwanted errors in our functions. If you are interested in learning more about the jaxpr representation, check out the JAX [documentation](#) on it. But for this tutorial, we just need the basics as discussed above.

Automatic differentiation

The intermediate jaxpr representation defines a computation graph, on which we can perform an essential operation of deep learning framework: automatic differentiation. In frameworks like PyTorch with a dynamic computation graph, we would compute the gradients based on the loss tensor itself, e.g. by calling `loss.backward()`. However, JAX directly works with functions. Instead of backpropagating gradients through tensors, JAX takes as input a function, and outputs another function which directly calculates the gradients for it. While this might seem quite different to what you are used to from other frameworks, it is quite intuitive: your gradient of parameters is really a function of parameters and data.

The transformation that allows us to do this is `jax.grad`, which takes as input the function, and returns another function representing the gradient calculation of the (first) input with respect to the output:

```
[18]: grad_function = jax.grad(simple_graph)
gradients = grad_function(inp)
print('Gradient', gradients)

Gradient [1.3333334 2.           2.6666667]
```

The gradient we get here is exactly the one we would obtain when doing the calculation by hand. Moreover, we can also print the jaxpr representation of the gradient function:

```
[19]: jax.make_jaxpr(grad_function)(inp)

[19]: { lambda ; a:f32[3]. let
    b:f32[3] = add a 2.0
    c:f32[3] = integer_pow[y=2] b
    d:f32[3] = integer_pow[y=1] b
    e:f32[3] = mul 2.0 d
    f:f32[3] = add c 3.0
    g:f32[] = reduce_sum[axes=(0,)] f
    _:f32[] = div g 3.0
    h:f32[] = div 1.0 3.0
    i:f32[3] = broadcast_in_dim[broadcast_dimensions=() shape=(3,)] h
    j:f32[3] = mul i e
    in (j,) }
```

This shows an unique property of JAX: we can print out the exact computation graph for determining the gradients. Compared to the original function, you can see new equations like `d:f32[3] = integer_pow[y=1] b` and `e:f32[3] = mul 2.0 d`, which model the intermediate gradient of $\partial b_i^2 / \partial b_i = 2b_i$. Furthermore, the return value `j` is the multiplication of `e` with $1/3$, which maps to the gradient being:

$$\frac{\partial y}{\partial x_i} = \frac{2}{3}(x_i + 2)$$

Hence, we can not only use JAX to estimate the gradients at a certain input value, but actually return the analytical gradient function which is quite a nice feature and highlights the properties of JAX!

Often, we do not only want the gradients, but also the actual output of the function, for instance for logging the loss. This can be efficiently done using `jax.value_and_grad`:

```
[20]: val_grad_function = jax.value_and_grad(simple_graph)
val_grad_function(inp)

[20]: (DeviceArray(12.666667, dtype=float32),
       DeviceArray([1.3333334, 2.           , 2.6666667], dtype=float32))
```

Further, we can specialize the gradient function to consider multiple input arguments, and add extra outputs that we may not want to differentiate (for instance the accuracy in classification). We will visit the most important ones in the network training later in this section, and refer to other great resources for more details ([JAX Quickstart](#), [Autodiff cookbook](#), [Advanced autodiff](#)).

To train neural networks, we need to determine the gradient for every parameter in the network with respect to the loss. Listing all parameters as input arguments quickly gets annoying and infeasible. JAX offers an elegant data structure to summarize all parameters: a pytree ([documentation](#)). A pytree is a container-like object which structures its elements as a tree. For instance, a linear neural network might have its parameters organized similar to:

```
params = {
    'linear1': {
        'weights': ...,
        'bias': ...
    },
    ...
}
```

JAX offers functions to process pytrees efficiently, such as obtaining all leafs (i.e. all parameters in a network) or applying a function on each element. We will come back to these structures when training a full network.

Speeding up computation with Just-In-Time compilation

Interestingly, from the previous code cell, you can see in the jaxpr representation of the gradient function that calculating the array `f` and scalar `g` are unnecessary. Intuitively, the gradient of taking the mean is independent of the actual output of the mean, hence we could drop `f` and `g` without any drawback. Finding such cases to improve efficiency and optimizing the code to take full advantage of the available accelerator hardware is one of the big selling points of JAX. It achieves that by *compiling functions just-in-time* with [XLA](#) (Accelerated Linear Algebra), using their jaxpr representation. Thereby, XLA fuses operations to reduce execution time of short-lived operations and eliminates intermediate storage buffers where not needed. For more details, see the [XLA documentation](#).

To compile a function, JAX provides the `jax.jit` transformation. We can either apply the transformation directly on a function (as we will do in the next cell), or use the decorator `@jax.jit` before a function.

```
[21]: jitted_function = jax.jit(simple_graph)
```

The `jitted_function` takes the same input arguments as the original function `simple_graph`. Since the jaxpr representation of a function depends on the input shape, the compilation is started once we put the first input in. However, note that this also means that for every different shape we want to run the function, a new XLA compilation is needed. This is why it is recommended to use padding in cases where your input shape strongly varies (we revisit this topic in the final section of this tutorial). For now, let's create an array with 1000 random values, on which we apply the jitted function:

```
[22]: # Create a new random subkey for generating new random values
rng, normal_rng = jax.random.split(rng)
large_input = jax.random.normal(normal_rng, (1000,))
# Run the jitted function once to start compilation
_ = jitted_function(large_input)
```

The output is not any different from what you would get from the non-jitted function. However, how is it about the runtime? Let's time both the original and the jitted function. Due to the asynchronous execution on the GPU, we add `.block_until_ready()` on the output, which blocks the Python execution until the accelerator (here GPU) finished computing the result and hence get an accurate time estimate.

```
[23]: %%timeit  
simple_graph(large_input).block_until_ready()  
598 µs ± 104 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
[24]: %%timeit  
jitted_function(large_input).block_until_ready()  
19.5 µs ± 52.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

We see that the compiled function is almost 10-15x faster! This is quite an improvement in performance, and shows the potential of compiling functions with XLA. Furthermore, we can also apply multiple transformations on the same function in JAX, such as applying `jax.jit` on a gradient function:

```
[25]: jitted_grad_function = jax.jit(grad_function)  
_ = jitted_grad_function(large_input) # Apply once to compile
```

Let's time the functions once more:

```
[26]: %%timeit  
grad_function(large_input).block_until_ready()  
2.55 ms ± 190 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
[27]: %%timeit  
jitted_grad_function(large_input).block_until_ready()  
17.4 µs ± 60.6 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Once more, the jitted function is much faster than the original one. Intuitively, this shows the potential speed up we can gain by using `jax.jit` to compile the whole training step of a network. Generally, we want to jit the largest possible chunk of computation to give the compiler maximum freedom.

There are situations in which applying `jit` to a function is not straight-forward, for instance, if an input argument cannot be traced, or you need to use loops that depend on input arguments. To keep the tutorial simple, and since most neural network training functions do not run into these issues, we do not discuss such special cases here. Instead, we refer to the section on just-in-time compilation in the great tutorials of [JAX 101 Tutorial](#), [JAX Quickstart](#), and [Thinking in JAX](#).

4.19.3 Implementing a Neural Network with Flax

With having reviewed the basics of JAX, we are now ready to implement our own neural network. Technically, we could implement our own neural network from scratch with JAX (see [here](#) for an example), but we do not really want to do that every time. Similarly to PyTorch's `torch.nn` package, there exist neural network libraries based on JAX which provide such basic functionality. A (non-exclusive) collection of them are:

- [Flax](#), started by the Google Brain Team, focuses on flexibility and clarity.
- [Haiku](#), from DeepMind, focuses on simplicity and compositionality.
- [Trax](#), maintained by the Google Brain Team, provides solutions for common training tasks
- [Equinox](#), created by Patrick Kidger and Cristian Garcia, implements neural networks as callable PyTrees
- [Jraph](#), from DeepMind, is a graph neural network library (similar to PyTorch Geometric)

For this tutorial series, we will use Flax due to its flexibility, intuitive API, and larger community. However, this should not mean that the other libraries are necessarily worse, and we recommend giving them a try as well to find the best library for yourself!

We will introduce the libraries and all additional parts you might need to train a neural network in Flax, using a simple example classifier on a simple yet well known example: XOR. Given two binary inputs x_1 and x_2 , the label to predict is 1 if either x_1 or x_2 is 1 while the other is 0, or the label is 0 in all other cases. The example became famous by the fact that a single neuron, i.e. a linear classifier, cannot learn this simple function. Hence, we will learn how to build a small neural network that can learn this function. To make it a little bit more interesting, we move the XOR into continuous space and introduce some gaussian noise on the binary inputs. Our desired separation of an XOR dataset could look as follows:

The model

The package `flax.linen` defines a series of useful classes like linear networks layers, convolutions, activation functions etc. A full list can be found [here](#). In case you need a certain network layer, check the documentation of the package first before writing the layer yourself as the package likely contains the code for it already. We import it below:

```
[28]: try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax

from flax import linen as nn
```

nn.Module

Similar to PyTorch, a neural network is built up out of modules. Modules can contain other modules, and a neural network is considered to be a module itself as well. The basic template of a module is as follows:

```
[29]: class MyModule(nn.Module):
    # Some dataclass attributes, like hidden dimension, number of layers, etc. of the
    # form:
    # varname : vartype

    def setup(self):
        # Flax uses "lazy" initialization. This function is called once before you
        # call the model, or try to access attributes. In here, define your submodules
        # etc.
        pass

    def __call__(self, x):
        # Function for performing the calculation of the module.
        pass
```

The main, obvious difference to PyTorch is that Flax uses lazy initialization. The function `setup` is called once on a module instance before any other methods are called, or when you try to access a attribute of `self` defined in `setup`. Additional object attributes are defined below the class name. However, in contrast to PyTorch, the parameters are not part of the module. Instead, we can create a set of parameters of the module by calling its `init()` function. This function takes as input a PRNG state for sampling pseudo-random numbers and an example input to the model, and

returns a set of parameters for the module as a pytree. Further, since the init function requires an input to the network, we can infer the input shape for all modules and do not need to explicitly define it during the module creation. This becomes more clear in the example we show in a second.

The `__call__` method represents the `forward` function in PyTorch, and performs the actual computation of the module. It can take additional arguments if needed, like whether we are training or validation.

Simple classifier

To get an intuition behind how we work with modules in Flax, let's define our own small neural network. We will use a minimal network with a input layer, one hidden layer with tanh as activation function, and a output layer. In other words, our networks should look something like this:

The input neurons are shown in blue, which represent the coordinates x_1 and x_2 of a data point. The hidden neurons including a tanh activation are shown in white, and the output neuron in red. In Flax, we can define this as follows:

```
[30]: class SimpleClassifier(nn.Module):
    num_hidden : int    # Number of hidden neurons
    num_outputs : int   # Number of output neurons

    def setup(self):
        # Create the modules we need to build the network
        # nn.Dense is a linear layer
        self.linear1 = nn.Dense(features=self.num_hidden)
        self.linear2 = nn.Dense(features=self.num_outputs)

    def __call__(self, x):
        # Perform the calculation of the model to determine the prediction
        x = self.linear1(x)
        x = nn.tanh(x)
        x = self.linear2(x)
        return x
```

One thing you may notice is that usually, all layers that we define in `setup` are also used in the `__call__` function. To reduce the code overhead, Flax provides an alternative, more compact network creation with `nn.compact`. With that, we can remove the `setup` function and instead our model as:

```
[31]: class SimpleClassifierCompact(nn.Module):
    num_hidden : int    # Number of hidden neurons
    num_outputs : int   # Number of output neurons

    @nn.compact # Tells Flax to look for defined submodules
    def __call__(self, x):
        # Perform the calculation of the model to determine the prediction
        # while defining necessary layers
        x = nn.Dense(features=self.num_hidden)(x)
        x = nn.tanh(x)
        x = nn.Dense(features=self.num_outputs)(x)
        return x
```

The `nn.compact` annotation of the `__call__` method signals Flax to look for submodules that we define in the forward pass. These are automatically recognized as such, so that we can use them for initialization etc. Which of the two model definition you use is often up to you (see the [Flax documentation](#) for some pros and cons for both methods). In the

following tutorials, we will mostly use the compact version, but occasionally come back to the explicit setup function where necessary. For instance, if we define more functions on a module besides `__call__` and want to reuse some modules, it is recommended to use the setup version.

For the examples in this notebook, we will use a tiny neural network with two input neurons and eight hidden neurons. As we perform binary classification, we will use a single output neuron. Note that we do not apply a sigmoid on the output yet. This is because other functions, especially the loss, are more efficient and precise to calculate on the original outputs instead of the sigmoid output. We will discuss the detailed reason later.

Now, let's create an instance of this network:

```
[32]: model = SimpleClassifier(num_hidden=8, num_outputs=1)
# Printing the model shows its attributes
print(model)

SimpleClassifier(
    # attributes
    num_hidden = 8
    num_outputs = 1
)
```

At this stage, the model has no parameters initialized. To do this, let's create a random input of our dataset, and apply the `init` function:

```
[33]: rng, inp_rng, init_rng = jax.random.split(rng, 3)
inp = jax.random.normal(inp_rng, (8, 2)) # Batch size 8, input size 2
# Initialize the model
params = model.init(init_rng, inp)
print(params)

FrozenDict({
    params: {
        linear1: {
            kernel: DeviceArray([[ 0.31476864, -0.4647768 , -0.7862042 , -0.48842615,
                                -0.65373844,  0.3892545 ,  0.3038056 ,  0.04179859],
                                [-0.3298236 ,  1.1110363 ,  0.54909396, -0.8168818 ,
                                0.40057245, -0.8665987 ,  1.2087964 ,  1.0364622 ]]),
            ← dtype=float32),
            bias: DeviceArray([0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32),
        },
        linear2: {
            kernel: DeviceArray([[-0.27971813],
                                [-0.7466775 ],
                                [ 0.29791608],
                                [-0.26267236],
                                [-0.5084385 ],
                                [ 0.04573093],
                                [-0.47257012],
                                [ 0.50458497]], dtype=float32),
            bias: DeviceArray([0.], dtype=float32),
        },
    },
})
```

Now, we have parameters with which we can apply the network. We see that the parameters follow the same structure as defined in our module, and each linear layer contains one kernel, i.e. the weights, and a bias parameter. With this, we could apply the model on an input using the `apply` function:

```
[34]: model.apply(params, inp)
[34]: DeviceArray([[-0.48368204],
                 [ 0.04365474],
                 [ 0.06668529],
                 [-0.34203646],
                 [ 0.4835147 ],
                 [ 0.37424874],
                 [ 0.14232653],
                 [-0.5916512 ]], dtype=float32)
```

The model returns an output array of shape `[8, 1]`, which corresponds to the one output neuron in the model for all 8 batch elements. With that, we now know how to initialize a model, and run a model. Next, let's look at the data.

The data

As mentioned before, JAX is not meant to ‘reinvent the wheel’ for every part of the deep learning pipeline. Hence, JAX and Flax do not natively provide a data loading functionality, but instead refer to other available libraries like Tensorflow and PyTorch. Here, let's use again the package `torch.utils.data` library.

```
[35]: import torch.utils.data as data
```

The data package defines two classes which are the standard interface for handling data in PyTorch: `data.Dataset`, and `data.DataLoader`. The dataset class provides an uniform interface to access the training/test data, while the data loader makes sure to efficiently load and stack the data points from the dataset into batches during training.

The dataset class

The dataset class summarizes the basic functionality of a dataset in a natural way. To define a dataset in PyTorch, we simply specify two functions: `__getitem__`, and `__len__`. The get-item function has to return the i -th data point in the dataset, while the len function returns the size of the dataset. For the XOR dataset, we can define the dataset class as follows:

```
[36]: class XORDataset(data.Dataset):
    def __init__(self, size, seed, std=0.1):
        """
        Inputs:
            size - Number of data points we want to generate
            seed - The seed to use to create the PRNG state with which we want to
        generate the data points
            std - Standard deviation of the noise (see generate_continuous_xor function)
        """
        super().__init__()
        self.size = size
        self.np_rng = np.random.RandomState(seed=seed)
        self.std = std
        self.generate_continuous_xor()

    def generate_continuous_xor(self):
        # Each data point in the XOR dataset has two variables, x and y, that can be
        either 0 or 1
```

(continues on next page)

(continued from previous page)

```

# The label is their XOR combination, i.e. 1 if only x or only y is 1 while the
# other is 0.
# If x=y, the label is 0.
data = self.np_rng.randint(low=0, high=2, size=(self.size, 2)).astype(np.float32)
label = (data.sum(axis=1) == 1).astype(np.int32)
# To make it slightly more challenging, we add a bit of gaussian noise to the
# data points.
data += self.np_rng.normal(loc=0.0, scale=self.std, size=data.shape)

self.data = data
self.label = label

def __len__(self):
    # Number of data point we have. Alternatively self.data.shape[0], or self.label.
    # shape[0]
    return self.size

def __getitem__(self, idx):
    # Return the idx-th data point of the dataset
    # If we have multiple things to return (data point and label), we can return
    # them as tuple
    data_point = self.data[idx]
    data_label = self.label[idx]
    return data_point, data_label

```

Note that in contrast to our PyTorch tutorial, we use NumPy to generate the random data. Similar to JAX, NumPy also allows the pseudo-number generation based on a PRNG state. Hence, for better reproducibility, we are doing the same here. Let's try to create such a dataset and inspect it:

```
[37]: dataset = XORDataset(size=200, seed=42)
print("Size of dataset:", len(dataset))
print("Data point 0:", dataset[0])

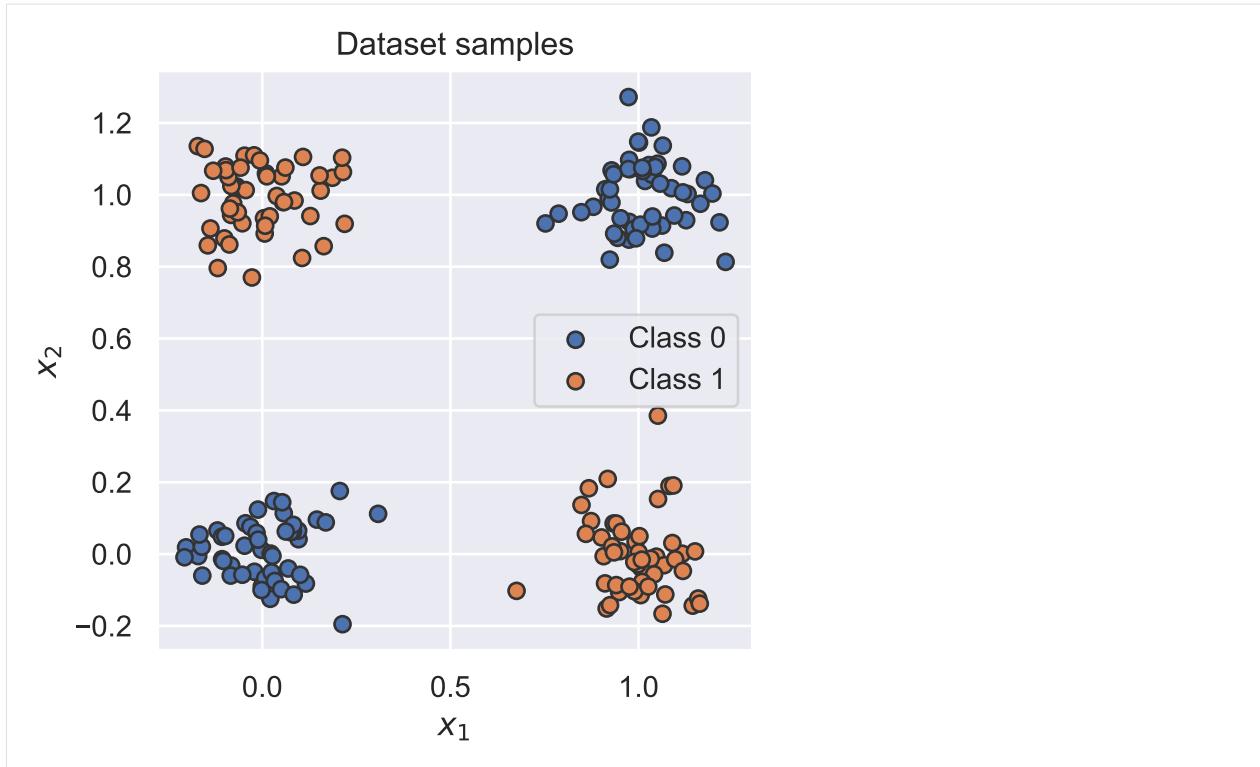
Size of dataset: 200
Data point 0: (array([-0.06800247,  1.0232254 ], dtype=float32), 1)
```

To better relate to the dataset, we visualize the samples below.

```
[38]: def visualize_samples(data, label):
    data_0 = data[label == 0]
    data_1 = data[label == 1]

    plt.figure(figsize=(4,4))
    plt.scatter(data_0[:,0], data_0[:,1], edgecolor="#333", label="Class 0")
    plt.scatter(data_1[:,0], data_1[:,1], edgecolor="#333", label="Class 1")
    plt.title("Dataset samples")
    plt.ylabel(r"$x_2$")
    plt.xlabel(r"$x_1$")
    plt.legend()
```

```
[39]: visualize_samples(dataset.data, dataset.label)
plt.show()
```



The data loader class

The class `torch.utils.data.DataLoader` represents a Python iterable over a dataset with support for automatic batching, multi-process data loading and many more features. The data loader communicates with the dataset using the function `__getitem__`, and stacks its outputs as tensors over the first dimension to form a batch. In contrast to the dataset class, we usually don't have to define our own data loader class, but can just create an object of it with the dataset as input. Additionally, we can configure our data loader with the following input arguments (only a selection, see full list [here](#)):

- `batch_size`: Number of samples to stack per batch
- `shuffle`: If True, the data is returned in a random order. This is important during training for introducing stochasticity.
- `num_workers`: Number of subprocesses to use for data loading. The default, 0, means that the data will be loaded in the main process which can slow down training for datasets where loading a data point takes a considerable amount of time (e.g. large images). More workers are recommended for those, but can cause issues on Windows computers. For tiny datasets as ours, 0 workers are usually faster.
- `persistent_workers`: If True, workers will not be shutdown after an iteration over the dataset has finished. This can be useful if the time per epoch is small, or if you face issues with workers being killed during training.
- `drop_last`: If True, the last batch is dropped in case it is smaller than the specified batch size. This occurs when the dataset size is not a multiple of the batch size. Only potentially helpful during training to keep a consistent batch size.
- `collate_fn`: A function that defines how the elements per batch are combined. By default, PyTorch stacks them as PyTorch tensors. For JAX, we will change it to NumPy arrays.

Let's create a simple data loader below with a function that stacks batch elements as NumPy array instead of PyTorch Tensors:

```
[40]: # This collate function is taken from the JAX tutorial with PyTorch Data Loading
# https://jax.readthedocs.io/en/latest/notebooks/Neural_Network_and_Data>Loading.html
def numpy_collate(batch):
    if isinstance(batch[0], np.ndarray):
        return np.stack(batch)
    elif isinstance(batch[0], (tuple, list)):
        transposed = zip(*batch)
        return [numpy_collate(samples) for samples in transposed]
    else:
        return np.array(batch)

data_loader = data.DataLoader(dataset, batch_size=8, shuffle=True, collate_fn=numpy_
→collate)
```

```
[41]: # next(iter(...)) catches the first batch of the data loader
# If shuffle is True, this will return a different batch every time we run this cell
# For iterating over the whole dataset, we can simply use "for batch in data_loader: ..."
data_inputs, data_labels = next(iter(data_loader))

# The shape of the outputs are [batch_size, d_1,...,d_N] where d_1,...,d_N are the
# dimensions of the data point returned from the dataset class
print("Data inputs", data_inputs.shape, "\n", data_inputs)
print("Data labels", data_labels.shape, "\n", data_labels)

Data inputs (8, 2)
[[ 1.0504987  1.0865755 ]
 [ 0.02809919 -0.06226995]
 [ 0.06141667  1.0757508 ]
 [ 0.08356921 -0.11297069]
 [ 1.0324166 -0.01301431]
 [ 1.0024511   0.04979983]
 [ 0.3078881   0.11195749]
 [ 1.0371146   0.9396015 ]]
Data labels (8,)
[0 0 1 0 1 1 0 0]
```

Optimization

After defining the model and the dataset, it is time to prepare the optimization of the model. During training, we will perform the following steps:

1. Get a batch from the data loader
2. Obtain the predictions from the model for the batch
3. Calculate the loss based on the difference between predictions and labels
4. Backpropagation: calculate the gradients for every parameter with respect to the loss
5. Update the parameters of the model in the direction of the gradients

We have seen how we can do step 1, 2 and 4 in JAX and Flax. Now, we will look at step 3 and 5.

Stochastic Gradient Descent

For updating the parameters, Flax does not directly provide support for optimizers, but instead refers to another package called `optax` ([documentation](#)). Optax is an optimization library for JAX, which offers most common deep learning optimizers (SGD, Adam, Adagrad, RMSProp, etc.) and utilities (gradient clipping, weight decay, etc.).

```
[42]: try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax
```

For now, we will use the simplest optimizer, namely `optax.sgd`. Stochastic Gradient Descent updates parameters by multiplying the gradients with a small constant, called learning rate, and subtracting those from the parameters (hence minimizing the loss). Therefore, we slowly move towards the direction of minimizing the loss. A good default value of the learning rate for a small network as ours is 0.1. Remember that we again aim to write functional code. Hence, the optimizer does not take as input the parameters, but only the optimizer hyperparameters.

```
[43]: # Input to the optimizer are optimizer settings like learning rate
optimizer = optax.sgd(learning_rate=0.1)
```

Since JAX calculates gradients via function transformations, we do not have functions like `backward()`, `optimizer.step()` or `optimizer.backward()` as in PyTorch. Instead, a optimizer is a function on the parameters and gradients. To simplify this step and bundle important parts of the training procedure, Flax offers the `flax.training` package. As a first step, we can create a `TrainState` which bundles the parameters, the optimizer, and the forward step of the model:

```
[44]: from flax.training import train_state

model_state = train_state.TrainState.create(apply_fn=model.apply,
                                            params=params,
                                            tx=optimizer)
```

With this state object, it is easier to handle the training.

Loss function

For performing gradient updates, we need a function that can calculate the loss for a batch. Afterwards, we can apply JAX's gradient transformation to obtain a gradient function of it. In our setting, which is binary classification, we can use Binary Cross Entropy (BCE) which is defined as follows:

$$\mathcal{L}_{BCE} = - \sum_i [y_i \log x_i + (1 - y_i) \log(1 - x_i)]$$

where y are our labels, and x our predictions, both in the range of $[0, 1]$. Similar to PyTorch, Optax already provides a function for this: `optax.sigmoid_binary_cross_entropy(logits, labels)`. We calculate the loss on the logits instead of the sigmoid outputs for numerical stability. Let's write a function that takes as input a state (for the forward function), parameters, and a batch, and return the binary cross entropy loss and accuracy:

```
[45]: def calculate_loss_acc(state, params, batch):
    data_input, labels = batch
    # Obtain the logits and predictions of the model for the input data
    logits = state.apply_fn(params, data_input).squeeze(axis=-1)
```

(continues on next page)

(continued from previous page)

```

pred_labels = (logits > 0).astype(jnp.float32)
# Calculate the loss and accuracy
loss = optax.sigmoid_binary_cross_entropy(logits, labels).mean()
acc = (pred_labels == labels).mean()
return loss, acc

```

Note that we explicitly add the parameters here as an input argument since we want to calculate the gradients with respect to them later. An example execution of the function would look like:

```
[46]: batch = next(iterator(data_loader))
calculate_loss_acc(model_state, model_state.params, batch)

[46]: (DeviceArray(0.6830494, dtype=float32), DeviceArray(0.625, dtype=float32))
```

Creating an efficient training and validation step

With this loss function and the optimizer, we are now ready to create an efficient training and validation/test step. First, let's consider the training. As input to each training step, we have a training state and a batch. We then want to calculate the loss for the input and take the gradients of it. Finally, we update the parameters with our optimizer and return the new state. All this can be summarized in the following function:

```
[47]: @jax.jit # Jit the function for efficiency
def train_step(state, batch):
    # Gradient function
    grad_fn = jax.value_and_grad(calculate_loss_acc, # Function to calculate the loss
                                 argnums=1, # Parameters are second argument of the
                                 # function
                                 has_aux=True # Function has additional outputs, here
                                 # accuracy
                                 )
    # Determine gradients for current model, parameters and batch
    (loss, acc), grads = grad_fn(state, state.params, batch)
    # Perform parameter update with gradients and optimizer
    state = state.apply_gradients(grads=grads)
    # Return state and any other value we might want
    return state, loss, acc
```

By using the transformation `jax.jit`, the whole gradient calculation and application is optimized in XLA, providing an efficient function for updating the model.

Next, let's look at the evaluation function. Here, we do not need to calculate gradients, but only want to get the accuracy of the model for the batch. This becomes a simpler version of the training step:

```
[48]: @jax.jit # Jit the function for efficiency
def eval_step(state, batch):
    # Determine the accuracy
    _, acc = calculate_loss_acc(state, state.params, batch)
    return acc
```

These two functions provide us now efficient utilities to train our model.

Training

Finally, we are ready to train our model. As a first step, we create a slightly larger dataset and specify a data loader with a larger batch size.

```
[49]: train_dataset = XORDataset(size=2500, seed=42)
train_data_loader = data.DataLoader(train_dataset, batch_size=128, shuffle=True, collate_fn=numpy_collate)
```

Now, we can write a small training function. In contrast to PyTorch, we do not need to explicitly push our model to GPU, since the parameters are already automatically created on GPU. Further, since the model itself is stateless, we do not have a `train()` or `eval()` function to switch between modes of e.g. dropout. When necessary, we can add an argument `train : bool` to the model forward pass. For this simple network here, however, this is not necessary.

Following the PyTorch tutorial, let's write a function here that trains a model for several epochs:

```
[50]: def train_model(state, data_loader, num_epochs=100):
    # Training loop
    for epoch in tqdm(range(num_epochs)):
        for batch in data_loader:
            state, loss, acc = train_step(state, batch)
            # We could use the loss and accuracy for logging here, e.g. in TensorBoard
            # For simplicity, we skip this part here
    return state
```

```
[51]: trained_model_state = train_model(model_state, train_data_loader, num_epochs=100)
0% | 0/100 [00:00<?, ?it/s]
```

Training this model for 100 epochs does take barely a second... This shows the impressive speed JAX can reach, especially for small models!

Saving a model

After we finished training a model, we save the model to disk so that we can load the same weights at a later time. In JAX, this means we want to save the `state.params` dictionary. Luckily, the `flax.training` package again provides us with nice utilities for that, which uses TensorFlow as underlying framework.

```
[52]: from flax.training import checkpoints
```

To save the whole model state, we can write:

```
[53]: checkpoints.save_checkpoint(ckpt_dir='my_checkpoints/', # Folder to save checkpoint in
                                target=trained_model_state, # What to save. To only save
                                # parameters, use model_state.params
                                step=100, # Training step or other metric to save best_
                                # model on
                                prefix='my_model', # Checkpoint file name prefix
                                overwrite=True # Overwrite existing checkpoint files
                               )
[53]: 'my_checkpoints/my_model100'
```

To load this state dict again, we can use `checkpoints.restore_checkpoint`:

```
[54]: loaded_model_state = checkpoints.restore_checkpoint(
    ckpt_dir='my_checkpoints/',    # Folder with_
    ↵the checkpoints
    ↵object to rebuild state in
    ↵prefix
    target=model_state,    # (optional) matching_
    prefix='my_model'  # Checkpoint file name_
)
```

The states `loaded_model_state` and `trained_model_state` have the identical parameter values.

Evaluation

Once we have trained a model, it is time to evaluate it on a held-out test set. As our dataset consist of randomly generated data points, we need to first create a test set with a corresponding data loader.

```
[55]: test_dataset = XORDataset(size=500, seed=123)
# drop_last -> Don't drop the last batch although it is smaller than 128
test_data_loader = data.DataLoader(test_dataset,
                                    batch_size=128,
                                    shuffle=False,
                                    drop_last=False,
                                    collate_fn=numpy_collate)
```

We can use our `eval_step` function to efficiently evaluate our model:

```
[56]: def eval_model(state, data_loader):
    all_accs, batch_sizes = [], []
    for batch in data_loader:
        batch_acc = eval_step(state, batch)
        all_accs.append(batch_acc)
        batch_sizes.append(batch[0].shape[0])
    # Weighted average since some batches might be smaller
    acc = sum([a*b for a,b in zip(all_accs, batch_sizes)]) / sum(batch_sizes)
    print(f"Accuracy of the model: {100.0*acc:4.2f}%")
```

```
[57]: eval_model(trained_model_state, test_data_loader)
```

```
Accuracy of the model: 100.00%
```

If we trained our model correctly, we should see a score close to 100% accuracy. However, this is only possible because of our simple task, and unfortunately, we usually don't get such high scores on test sets of more complex tasks.

Binding model parameters

Once we have trained the model, we might want to do multiple application of the same model and parameters. It can get a bit annoying to always write `model.apply(params, ...)` and keep track of the model and parameters separately. To prevent this, Flax's module can be bound to specific parameters to simplify our application. Specifically, we can bind the instance `model` of our `SimpleClassifier` class to our trained parameter as follows:

```
[58]: trained_model = model.bind(trained_model_state.params)
```

With the model being binded to the parameters, we can use it as we would any PyTorch module. For instance, to apply it to an input array, we can simply run:

```
[59]: data_input, labels = next(iter(data_loader))
out = trained_model(data_input) # No explicit parameter passing necessary anymore
out.shape
```



```
[59]: (8, 1)
```

This can simplify the analysis of models, and provide a more familiar interface to PyTorch users.

Visualizing classification boundaries

To visualize what our model has learned, we can perform a prediction for every data point in a range of $[-0.5, 1.5]$, and visualize the predicted class as in the sample figure at the beginning of this section. This shows where the model has created decision boundaries, and which points would be classified as 0, and which as 1. We therefore get a background image out of blue (class 0) and orange (class 1). The spots where the model is uncertain we will see a blurry overlap. The specific code is less relevant compared to the output figure which should hopefully show us a clear separation of classes:

```
[60]: def visualize_classification(model, data, label):
    data_0 = data[label == 0]
    data_1 = data[label == 1]

    fig = plt.figure(figsize=(4,4), dpi=500)
    plt.scatter(data_0[:,0], data_0[:,1], edgecolor="#333", label="Class 0")
    plt.scatter(data_1[:,0], data_1[:,1], edgecolor="#333", label="Class 1")
    plt.title("Dataset samples")
    plt.ylabel(r"$x_2$")
    plt.xlabel(r"$x_1$")
    plt.legend()

    # Let's make use of a lot of operations we have learned above
    c0 = np.array(to_rgba("C0"))
    c1 = np.array(to_rgba("C1"))
    x1 = jnp.arange(-0.5, 1.5, step=0.01)
    x2 = jnp.arange(-0.5, 1.5, step=0.01)
    xx1, xx2 = jnp.meshgrid(x1, x2, indexing='ij') # Meshgrid function as in numpy
    model_inputs = np.stack([xx1, xx2], axis=-1)
    logits = model(model_inputs)
    preds = nn.sigmoid(logits)
    output_image = (1 - preds) * c0[None, None] + preds * c1[None, None] # Specifying
    ↵ "None" in a dimension creates a new one
    output_image = jax.device_get(output_image) # Convert to numpy array. This only
    ↵ works for tensors on CPU, hence first push to CPU
```

(continues on next page)

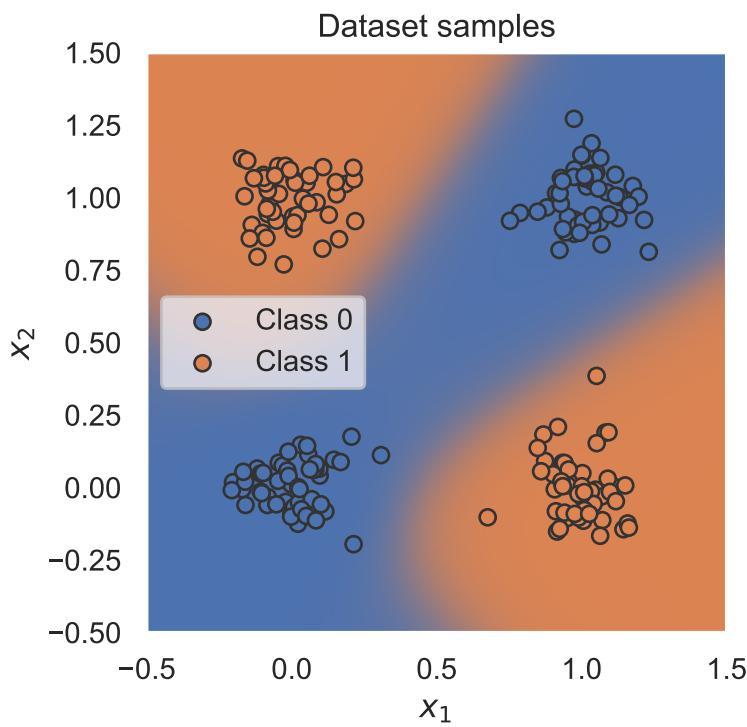
(continued from previous page)

```

plt.imshow(output_image, origin='lower', extent=(-0.5, 1.5, -0.5, 1.5))
plt.grid(False)
return fig

_ = visualize_classification(trained_model, dataset.data, dataset.label)
plt.show()

```



The decision boundaries might not look exactly as in the figure in the preamble of this section, since this has been created with the PyTorch version of the tutorial. Nevertheless, the result on the accuracy metric should be the approximately the same.

Conclusion

This concludes our tutorial on training a neural network with JAX. While the functional programming perspective of JAX may seem very different to PyTorch at first, it enables a considerable speedup in training, not only for tiny models like here. If you are interested in seeing more practical use cases of JAX, we recommend checking out our other JAX Tutorials, such as:

- [Tutorial 5 \(JAX\): Inception, ResNet and DenseNet](#) gives an intro to training convolutional classifiers on CIFAR10;
- [Tutorial 6 \(JAX\): Transformers and Multi-Head Attention](#) builds a Transformer from scratch with Flax;
- [Tutorial 7 \(JAX\): Graph Neural Networks](#) implements basic Graph Neural Network layers;
- [Tutorial 9 \(JAX\): Deep Autoencoders](#) shows how to train autoencoders on CIFAR10;
- [Tutorial 11 \(JAX\): Normalizing Flows for image modeling](#) discusses Normalizing Flows as generative model on images;
- [Tutorial 15 \(JAX\): Vision Transformers](#) trains a Transformer on image classification for CIFAR10.

4.19.4 The Fancy Bits

After reading this tutorial, you might wonder why we left out some key advertisement points of JAX: automatic vectorization, easy parallelization on multiple accelerators, etc. The reason why we did not include them in our previous discuss is that for building simple networks, and actual most models in our tutorials, you do not really need these methods. However, since they can be handy at some times, for instance, if you have access to a large cluster or are faced with functions that are annoying to vectorize, we review them here in a separate section: the Fancy Bits of JAX (the title is inspired by JAX's tutorial [JAX - The Sharp Bits](#)).

Automatic Vectorization with `vmap`

In machine learning, we often vectorize methods to efficiently process multiple inputs or batch elements at the same time. Usually, we have to write the code ourselves to support additional dimensions to vectorize over. However, since JAX can already transform functions to run efficiently on accelerators or calculate gradients, it can also automatically vectorize a function. For instance, let's consider a simple linear layer where we write a function for a single input `x` of shape `[c_in]`, a weight matrix `[c_in, c_out]`, and a bias vector `[c_out]`:

```
[61]: def simple_linear(x, w, b):
    # We could already vectorize this function with matmul, but as an example,
    # let us use a non-vectorized function with same output
    return (x[:,None] * w).sum(axis=0) + b
```

```
[62]: # Example inputs
rng, x_rng, w_rng, b_rng = jax.random.split(rng, 4)
x_in = jax.random.normal(x_rng, (4,))
w_in = jax.random.normal(w_rng, (4, 3))
b_in = jax.random.normal(b_rng, (3,))

simple_linear(x_in, w_in, b_in)
```

```
[62]: DeviceArray([-0.5393317,  1.4906642,  0.7108946], dtype=float32)
```

Now, we would like the function to support a batch dimension on `x`, i.e. `[batch, c_in]`. Our naive implementation above does not support this, since we specialized the axis we sum over. So, let's make JAX do the work for us and vectorize the function by using `jax.vmap`:

```
[63]: vectorized_linear = jax.vmap(simple_linear,
                                  in_axes=(0, None, None), # Which axes to vectorize for
                                  # each input
                                  out_axes=0 # Which axes to map to in the output
                                )
```

Specifying `None` for the in-axes of the input arguments `w` and `b` means that we do not want to vectorize any of their input dimensions. With this `vmap` specification, the function `vectorized_linear` now supports an extra batch dimension in `x`! Let's try it out:

```
[64]: x_vec_in = jnp.stack([x_in]*5, axis=0)

vectorized_linear(x_vec_in, w_in, b_in)

[64]: DeviceArray([[ -0.5393317,   1.4906642,   0.7108946],
                 [-0.5393317,   1.4906642,   0.7108946],
                 [-0.5393317,   1.4906642,   0.7108946],
```

(continues on next page)

(continued from previous page)

```
[ -0.5393317,  1.4906642,  0.7108946],  
[ -0.5393317,  1.4906642,  0.7108946]], dtype=float32)
```

The new function indeed vectorized our code, calculating N applications of the weights and bias to the input. We can also vectorize the code to run multiple inputs \mathbf{x} on multiple weights \mathbf{w} and biases \mathbf{b} by changing the input argument `in_axes` to `(0, 0, 0)`, or simply `0`. Moreover, we can again stack multiple function transformations, such as jitting a vectorized function. Further details on `jax.vmap` can be found in this [tutorial](#) and its [documentation](#).

Parallel evaluation with `pmap`

`jax.vmap` vectorizes a function on a single accelerator. But what if we have multiple GPUs or TPUs available? In PyTorch, we can parallelize a model over multiple GPUs using `nn.DistributedDataParallel`. In JAX, this is yet another function transformation: `jax.pmap`. Similar to `jax.vmap`, we can specify over which axes each input should be parallelized. In a network training, we usually want to parallelize over an extra batch dimension in the data, while the parameters are identical for all devices. For more details on `jax.pmap`, see [Parallel Evaluation in JAX](#).

Working with PyTrees

Network parameters in Flax are stored in a PyTree. We have visited them before, but what we haven't discuss yet is JAX's utilities to operate on pytrees! One common application is to obtain a list of all parameters in the network. This corresponds to extracting all leafs from a PyTree, for which JAX provides the function `jax.tree_leaves`:

```
[65]: parameters = jax.tree_leaves(model_state.params)  
print('We have parameters with the following shapes:', ', '.join([str(p.shape) for p in  
    parameters]))  
print('Overall parameter count:', sum([np.prod(p.shape) for p in parameters]))  
  
We have parameters with the following shapes: (8,), (2, 8), (1,), (8, 1)  
Overall parameter count: 33
```

We can also create new PyTrees that are the result of applying a function on all elements in the tree using `jax.tree_map`. For instance, let's obtain a PyTree with all parameter shapes:

```
[66]: jax.tree_map(lambda p: p.shape, model_state.params)  
  
[66]: FrozenDict({  
    params: {  
        linear1: {  
            bias: (8,),  
            kernel: (2, 8),  
        },  
        linear2: {  
            bias: (1,),  
            kernel: (8, 1),  
        },  
    },  
})
```

The nodes of PyTrees do not necessarily need to be NumPy or JAX arrays, but can be arbitrary objects. Overall, PyTrees provide a simple, structured representation of data useful in many applications. More details can be found in JAX's Tutorial [Working with PyTrees](#).

4.19.5 The Sharp Bits

Since JAX functions need to be written with certain constraints, there are situations where this can get annoying or difficult. A great overview of those, why they are needed, and most importantly, what to do about them, can be found in the original JAX tutorial [JAX - The Sharp Bits](#). In this final section of the tutorial, we want to visit a few of those points we have not explicitly discussed yet. Furthermore, we also focus on the combination with Flax, and what can be annoying when training models.

Dynamic shapes

JAX has the great advantage of providing just-in-time compilation of functions to speed up the computation. For this, it uses its intermediate representation jaxpr, which is specialized to the shapes of the input arguments. However, this also means that a jitted function is specialized to a certain shape, and running the jitted function with a different input shape requires recompiling the function. For instance, consider the following simple function:

```
[67]: def my_function(x):
    print('Running the function with shape', x.shape)
    return x.mean()

jitted_function = jax.jit(my_function)
```

The print statement is only executed once when the function is compiled, and for all consecutive function calls, this print statement will be ignored since it is not part of the jaxpr representation. Let's run the function now with multiple different input shapes:

```
[68]: for i in range(10):
    jitted_function(jnp.zeros(i+1,))

Running the function with shape (1,)
Running the function with shape (2,)
Running the function with shape (3,)
Running the function with shape (4,)
Running the function with shape (5,)
Running the function with shape (6,)
Running the function with shape (7,)
Running the function with shape (8,)
Running the function with shape (9,)
Running the function with shape (10,)
```

As we can see, the function is compiled for every different input we give it. This can become inefficient if we actually work with many different shapes. However, running the function again with one of the previous input shapes will not require another compilation:

```
[69]: # Running the functions a second time will not print out anything since
# the functions are already jitted for the respective input shapes.
for i in range(10):
    jitted_function(jnp.zeros(i+1,))
```

If we have a very limited set of different shapes, we do not see a big performance difference. For instance, in our evaluation, the last batch size is smaller than the previous since we have a limited size of the evaluation dataset. However, for other applications, we might encounter this problem much more often: NLP and time series, and graphs. In these cases, it is recommend to pad the batches to prevent many re-compilations (see Flax's [padding guide](#) for details). We briefly review the two scenarios below.

NLP and time series

In Natural Language Processing, our data consist of sentences which greatly vary in size. Already for batching the elements, we need to apply padding, such that the shape of the batch is determined by the largest sentence in the batch. However, this largest length can vary between batches, especially when we shuffle the dataset before each epoch. In PyTorch, this is not a problem, since the dynamic computation graph allows us to stop the computation whenever we need to. In contrast, JAX would need to recompile the forward pass for every single largest sentence length, which can quickly become very expensive. Padding is needed to reduce the number of compilations, but at the same time introduces unnecessary computation. Hence, we have a tradeoff between number of compilations and extra compute per batch. In the extreme case, PyTorch may even become faster than JAX here.

Graphs

Similar to NLP, graphs can vary in their size. Often, graphs differ in their number of nodes, but especially in the number of edges. Furthermore, when we start batching the graphs, the variation of node sizes and edge count considerably increases. Again, padding is needed to reduce the number of compilations, and we will revisit this topic in Tutorial 7 (TBD).

Debugging in jitted functions

During coding, we likely want to debug our model and sometimes print out intermediate values. In JAX, when jitting functions, this is not as straightforward. As we could see from the previous cells, a print statement is only executed once during compilation, and afterwards removed since it is not part of the jaxpr representation. Furthermore, there can be issues when tracking NaNs in your code (see the [sharp bits tutorial](#)), and errors like out-of-bounds indexing are silently handled on accelerators by returning -1 instead of an error (see the corresponding section in the [sharp bits tutorial](#)). However, if necessary, one can either run the unjitted version of the forward pass first, and even introduce print statements to the jitted version where needed (see [here](#) for a great explanation). Still, it is not as straight-forward as in PyTorch, for example.

Modules with different train and evaluation functions

Certain deep learning layers have different behaviors under evaluation than during training. For instance, dropout randomly masks out a number of neurons during training, but leaves the graph untouched during evaluation. In PyTorch, we can easily switch between the two states via `model.train()` and `model.eval()` without having to manually specify it in the dropout module instance. However, in JAX, we do not have global states in the model, and thus need to pass this information to every module in the forward pass that may need it. In our example above, this was not needed, since the forward pass of the simple classifier is identical during training and evaluation. Alternatively, since the parameters are not bound to a specific model during training, one can also create two models: one training model, and one evaluation model. Nonetheless, one still needs to add the information to every module with changing behaviors, which adds a certain overhead compared to PyTorch. We will discuss two common modules with such behaviors below: dropout and BatchNorm.

Dropout

In Flax, [Dropout](#) has an argument `deterministic` which turns off dropout when `True`, and otherwise applies the random masking as intended during training. This deterministic switch can either be defined in the constructor, or in every forward call. Furthermore, dropout has the special case that it is a random operation during training, meaning that it also needs a PRNG state. Fortunately, we do not have to pass this state in every PRNG state, but instead, can simply pass `rngs={'dropout': dropout_rng}` with `dropout_rng` being the PRNG state. For an example, see our [Tutorial 6](#) on Transformers in which we use dropout in several occasions.

BatchNorm

Batch Normalization transforms the input in two different ways. During training, we determine the mean and standard deviation of the input, and normalize the data with it to a zero mean and standard deviation of one. During evaluation, on the other hand, we take a running statistic over the previous several batches, and use those to estimate the mean and standard deviation. This is necessary to keep the evaluation stable and invariant to the specific batches we choose. Still, in the Flax module ([documentation](#)), we need to give the argument `use_running_average` (`bool`) to either the constructor or each forward pass. Furthermore, BatchNorm has a specific property we haven't discussed yet and is a bit tricky in JAX: keeping track of the running average. During every forward pass, we want to record the mean and standard deviation of the current batch, and update our current average over the past batches. However, this requires changing an input tensor, and returning this changed tensor again. In Flax, we can do this by defining the batch statistics as a mutable tensor. Check out our [Tutorial 5](#) to see BatchNorm being used in practice with Flax.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on [GitHub](#).

4.20 Tutorial 3 (JAX): Activation Functions

Filled notebook:

Pre-trained models:

PyTorch version:

Author: Phillip Lippe

Note: This notebook is written in JAX+Flax. It is a 1-to-1 translation of the original notebook written in PyTorch+PyTorch Lightning with almost identical results. For an introduction to JAX, check out our [Tutorial 2 \(JAX\): Introduction to JAX+Flax](#). Further, throughout the notebook, we comment on major differences to the PyTorch version and provide explanations for the major parts of the JAX code. We do not provide speed comparisons for this notebook since they tend to be uninformative for such small networks and potentially bottlenecked by other factors.

In this tutorial, we will take a closer look at (popular) activation functions and investigate their effect on optimization properties in neural networks. Activation functions are a crucial part of deep learning models as they add the non-linearity to neural networks. There is a great variety of activation functions in the literature, and some are more beneficial than others. The goal of this tutorial is to show the importance of choosing a good activation function (and how to do so), and what problems might occur if we don't.

Before we start, we import our standard libraries and set up basic functions:

```
[1]: ## Standard libraries
import os
import json
import math
import numpy as np
from typing import Any, Sequence
import pickle
from copy import deepcopy

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
import seaborn as sns
sns.set()

## Progress bar
from tqdm.auto import tqdm

## JAX
import jax
import jax.numpy as jnp
from jax import random

## Flax (NN in JAX)
try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax
from flax import linen as nn
from flax.training import train_state, checkpoints

## Optax (Optimizers in JAX)
try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax
```

In contrast to PyTorch, we do not have to set the seed as a global variable since we do not allow for functions with side effects. Instead, we explicitly define a PRNG key whenever needed for random operations like network initializations.

Additionally, the following cell defines two paths: `DATASET_PATH` and `CHECKPOINT_PATH`. The dataset path is the directory where we will download datasets used in the notebooks. It is recommended to store all datasets from PyTorch in one joined directory to prevent duplicate downloads. The checkpoint path is the directory where we will store trained

model weights and additional files. The needed files will be automatically downloaded. In case you are on Google Colab, it is recommended to change the directories to start from the current directory (i.e. remove `..../` for both dataset and checkpoint path).

```
[2]: # Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../../../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../../../saved_models/tutorial3_jax"

# Verifying the device that will be used throughout this notebook
print("Device:", jax.devices()[0])

Device: gpu:0
```

The following cell downloads all pretrained models we will use in this notebook. The files are stored on a separate [repository](#) to reduce the size of the notebook repository, especially for building the documentation on ReadTheDocs.

```
[3]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/JAX/tutorial3/"
# Files to download
pretrained_files = ["FashionMNIST_elu.config", "FashionMNIST_elu.tar",
                    "FashionMNIST_leakyrelu.config", "FashionMNIST_leakyrelu.tar",
                    "FashionMNIST_relu.config", "FashionMNIST_relu.tar",
                    "FashionMNIST_sigmoid.config", "FashionMNIST_sigmoid.tar",
                    "FashionMNIST_swish.config", "FashionMNIST_swish.tar",
                    "FashionMNIST_tanh.config", "FashionMNIST_tanh.tar"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please contact the author with the full output.\nincluding the following error:\n", e)
```

4.20.1 Common activation functions

As a first step, we will implement some common activation functions by ourselves. Of course, most of them can also be found in the `flax.linen` or `jax.nn` package (see the [documentation](#) for an overview). However, we'll write our own functions here for a better understanding and insights.

Every activation function will be an `nn.Module` so that we can integrate them nicely in a network. We start with implementing two of the “oldest” activation functions that are still commonly used for various tasks: sigmoid and tanh. Both the sigmoid and tanh activation can be also found as JAX functions (`nn.sigmoid`, `nn.tanh`). Note that since all modules are essentially functions, you can directly use functions in `nn.Sequential` (e.g. `nn.Sequential([nn.Dense(8), nn.sigmoid, nn.Dense(1)])`). Still, for a better understanding, we implement them by hand:

```
[4]: #####
class Sigmoid(nn.Module):
    def __call__(self, x):
        return 1 / (1 + jnp.exp(-x))

#####
class Tanh(nn.Module):
    def __call__(self, x):
        x_exp, neg_x_exp = jnp.exp(x), jnp.exp(-x)
        return (x_exp - neg_x_exp) / (x_exp + neg_x_exp)

#####
```

Another popular activation function that has allowed the training of deeper networks, is the Rectified Linear Unit (ReLU). Despite its simplicity of being a piecewise linear function, ReLU has one major benefit compared to sigmoid and tanh: a strong, stable gradient for a large range of values. Based on this idea, a lot of variations of ReLU have been proposed, of which we will implement the following three: LeakyReLU, ELU, and Swish. LeakyReLU replaces the zero settings in the negative part with a smaller slope to allow gradients to flow also in this part of the input. Similarly, ELU replaces the negative part with an exponential decay. The third, most recently proposed activation function is Swish, which is actually the result of a large experiment with the purpose of finding the “optimal” activation function. Compared to the other activation functions, Swish is both smooth and non-monotonic (i.e. contains a change of sign in the gradient). This has been shown to prevent dead neurons as in standard ReLU activation, especially for deep networks. If interested, a more detailed discussion of the benefits of Swish can be found in [this paper](#) [1].

Let's implement the four activation functions below:

```
[5]: #####
class ReLU(nn.Module):
    def __call__(self, x):
        return jnp.maximum(x, 0)

#####
class LeakyReLU(nn.Module):
    alpha : float = 0.1

    def __call__(self, x):
        return jnp.where(x > 0, x, self.alpha * x)

#####
class ELU(nn.Module):

    def __call__(self, x):
        return jnp.where(x > 0, x, jnp.exp(x)-1)

#####
```

(continues on next page)

(continued from previous page)

```
class Swish(nn.Module):

    def __call__(self, x):
        return x * nn.sigmoid(x)

#####
```

For later usage, we summarize all our activation functions in a dictionary mapping the name to the class object. In case you implement a new activation function by yourself, add it here to include it in future comparisons as well:

```
[6]: act_fn_by_name = {
    "sigmoid": Sigmoid,
    "tanh": Tanh,
    "relu": ReLU,
    "leakyrelu": LeakyReLU,
    "elu": ELU,
    "swish": Swish
}
```

Visualizing activation functions

To get an idea of what each activation function actually does, we will visualize them in the following. Since modules are functions in disguise, we can do this by applying JAX's gradient transformation `jax.grad`. However, note that we want to take the gradients for multiple inputs independently and the gradient transformation requires the output of the forward pass to be a single scalar. We can implement this by simply summing the outputs of the activation function, since a sum distributes the gradients equally to all its inputs.

```
[7]: def get_grads(act_fn, x):
    """
    Computes the gradients of an activation function at specified positions.

    Inputs:
        act_fn - An module or function of the forward pass of the activation function.
        x - 1D input array.
    Output:
        An array with the same size of x containing the gradients of act_fn at x.
    """
    return jax.grad(lambda inp: act_fn(inp).sum()(x))
```

Now we can visualize all our activation functions including their gradients:

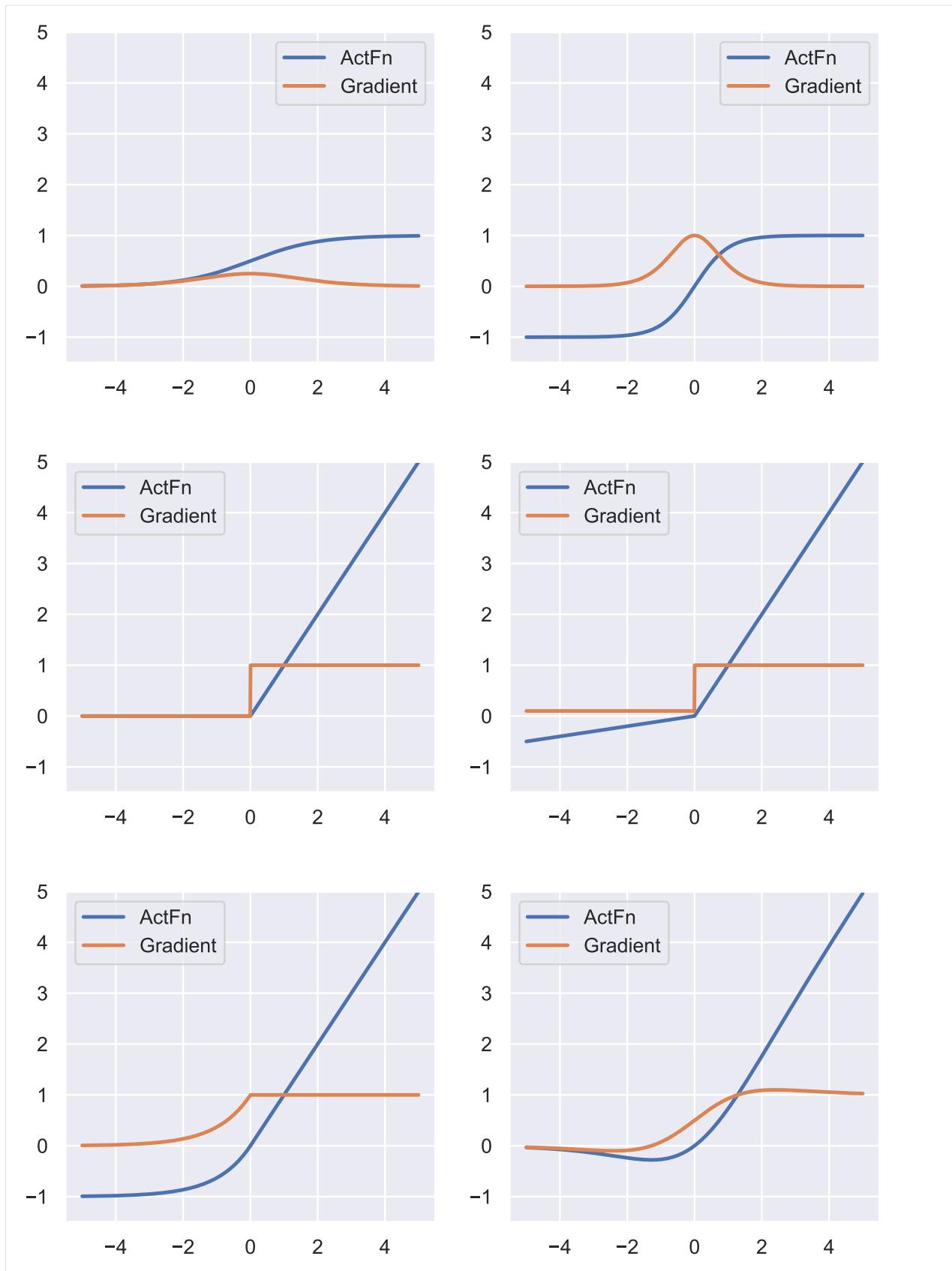
```
[8]: def vis_act_fn(act_fn, ax, x):
    # Run activation function
    y = act_fn(x)
    y_grads = get_grads(act_fn, x)
    # Push x, y and gradients back to cpu for plotting
    # x, y, y_grads = x.cpu().numpy(), y.cpu().numpy(), y_grads.cpu().numpy()
    ## Plotting
    ax.plot(x, y, linewidth=2, label="ActFn")
    ax.plot(x, y_grads, linewidth=2, label="Gradient")
```

(continues on next page)

(continued from previous page)

```
ax.set_title(act_fn.name)
ax.legend()
ax.set_ylim(-1.5, x.max())

# Add activation functions if wanted
act_fns = [act_fn() for act_fn in act_fn_by_name.values()]
x = np.linspace(-5, 5, 1000) # Range on which we want to visualize the activation
functions
## Plotting
rows = math.ceil(len(act_fns)/2.0)
fig, ax = plt.subplots(rows, 2, figsize=(8, rows*4))
for i, act_fn in enumerate(act_fns):
    vis_act_fn(act_fn, ax[divmod(i, 2)], x)
fig.subplots_adjust(hspace=0.3)
plt.show()
```



4.20.2 Analysing the effect of activation functions

After implementing and visualizing the activation functions, we are aiming to gain insights into their effect. We do this by using a simple neural network trained on [FashionMNIST](#) and examine various aspects of the model, including the performance and gradient flow.

Setup

Firstly, let's set up a neural network. The chosen network views the images as 1D tensors and pushes them through a sequence of linear layers and a specified activation function. Feel free to experiment with other network architectures. We explicitly set the initialization for the weights and biases to stay close to the [PyTorch implementation](#) of the Tutorial. As we will see in [Tutorial 4](#), the initialization has a considerable influence on which networks may be trainable with which activation functions.

```
[9]: # To keep the results close to the PyTorch tutorial, we use the same init function as
# PyTorch
# which is uniform(-1/sqrt(in_features), 1/sqrt(in_features)) - similar to He et al./
# kaiming
# The default for Flax is lecun_normal (i.e., half the variance of He) and zeros for
# bias.
init_func = lambda x: (lambda rng, shape, dtype: random.uniform(rng,
                                                               shape=shape,
                                                               minval=-1/np.sqrt(x.
                                                                     shape[1]),
                                                               maxval=1/np.sqrt(x.
                                                                     shape[1]),
                                                               dtype=dtype))

# Network
class BaseNetwork(nn.Module):
    act_fn : nn.Module
    num_classes : int = 10
    hidden_sizes : Sequence = (512, 256, 256, 128)

    @nn.compact
    def __call__(self, x, return_activations=False):
        x = x.reshape(x.shape[0], -1) # Reshape images to a flat vector
        # We collect all activations throughout the network for later visualizations
        # Remember that in jitted functions, unused tensors will anyways be removed.
        activations = []
        for hd in self.hidden_sizes:
            x = nn.Dense(hd,
                         kernel_init=init_func(x),
                         bias_init=init_func(x))(x)
            activations.append(x)
            x = self.act_fn(x)
            activations.append(x)
        x = nn.Dense(self.num_classes,
                     kernel_init=init_func(x),
                     bias_init=init_func(x))(x)
        return x if not return_activations else (x, activations)
```

We also add functions for loading and saving the model. The hyperparameters are stored in a configuration file (simple

json file):

```
[10]: def _get_config_file(model_path, model_name):
    # Name of the file for storing hyperparameter details
    return os.path.join(model_path, model_name + ".config")

def _get_model_file(model_path, model_name):
    # Name of the file for storing network parameters
    return os.path.join(model_path, model_name + ".tar")

def load_model(model_path, model_name, state=None):
    """
    Loads a saved model from disk.

    Inputs:
        model_path - Path of the checkpoint directory
        model_name - Name of the model (str)
        state - (Optional) If given, the parameters are loaded into this training state.
    ↪Otherwise,
                a new one is created alongside a network architecture.
    """

    config_file, model_file = _get_config_file(model_path, model_name), _get_model_file(model_path, model_name)
    assert os.path.isfile(config_file), f"Could not find the config file \'{config_file}\'. Are you sure this is the correct path and you have your model config stored here?"
    assert os.path.isfile(model_file), f"Could not find the model file \'{model_file}\'. Are you sure this is the correct path and you have your model stored here?"
    with open(config_file, "r") as f:
        config_dict = json.load(f)
    if state is None:
        act_fn_name = config_dict["act_fn"].pop("name").lower()
        act_fn = act_fn_by_name[act_fn_name](**config_dict.pop("act_fn"))
        net = BaseNetwork(act_fn=act_fn, **config_dict)
        state = train_state.TrainState(step=0,
                                        params=None,
                                        apply_fn=net.apply,
                                        tx=None,
                                        opt_state=None)
    else:
        net = None
    # You can also use flax's checkpoint package. To show an alternative,
    # you can instead load the parameters simply from a pickle file.
    with open(model_file, 'rb') as f:
        params = pickle.load(f)
        state = state.replace(params=params)
    return state, net

def save_model(model, params, model_path, model_name):
    """
    Given a model, we save the parameters and hyperparameters.

    Inputs:
        model - Network object without parameters
    """

    # Implementation of save_model function
```

(continues on next page)

(continued from previous page)

```

params - Parameters to save of the model
model_path - Path of the checkpoint directory
model_name - Name of the model (str)

"""
config_dict = deepcopy(model.__dict__)
config_dict['act_fn'] = config_dict['act_fn'].__dict__
for k in ['parent', 'name', '_state']:
    config_dict.pop(k)
config_dict['act_fn'].pop(k)
config_dict['act_fn']['name'] = model.act_fn.__class__.__name__.lower()
os.makedirs(model_path, exist_ok=True)
config_file, model_file = _get_config_file(model_path, model_name), _get_model_
file(model_path, model_name)
with open(config_file, "w") as f:
    json.dump(config_dict, f)
# You can also use flax's checkpoint package. To show an alternative,
# you can instead save the parameters simply in a pickle file.
with open(model_file, 'wb') as f:
    pickle.dump(params, f)

```

We also set up the dataset we want to train it on, namely [FashionMNIST](#). FashionMNIST is a more complex version of MNIST and contains black-and-white images of clothes instead of digits. The 10 classes include trousers, coats, shoes, bags and more. To load this dataset, we will make use of yet another PyTorch package, namely [torchvision](#) ([documentation](#)). The [torchvision](#) package consists of popular datasets, model architectures, and common image transformations for computer vision. We will use the package for many of the notebooks in this course to simplify our dataset handling.

Let's load the dataset below, and visualize a few images to get an impression of the data.

```
[11]: import torch
import torch.utils.data as data
import torchvision
from torchvision.datasets import FashionMNIST
from torchvision import transforms

# Transformations applied on each image => bring them into a numpy array and normalize_
# between -1 and 1
def image_to_numpy(img):
    img = np.array(img, dtype=np.float32)
    img = (img / 255. - 0.5) / 0.5
    return img

# We need to stack the batch elements as numpy arrays
def numpy_collate(batch):
    if isinstance(batch[0], np.ndarray):
        return np.stack(batch)
    elif isinstance(batch[0], (tuple, list)):
        transposed = zip(*batch)
        return [numpy_collate(samples) for samples in transposed]
    else:
        return np.array(batch)

# Loading the training dataset. We need to split it into a training and validation part
```

(continues on next page)

(continued from previous page)

```

train_dataset = FashionMNIST(root=DATASET_PATH,
                             train=True,
                             transform=image_to_numpy,
                             download=True)
train_set, val_set = torch.utils.data.random_split(train_dataset,
                                                   [50000, 10000],
                                                   generator=torch.Generator().manual_
                                                   ←seed(42))

# Loading the test set
test_set = FashionMNIST(root=DATASET_PATH,
                        train=False,
                        transform=image_to_numpy,
                        download=True)

# We define a set of data loaders that we can use for various purposes later.
# Note that for actually training a model, we will use different data loaders
# with a lower batch size.
train_loader = data.DataLoader(train_set,
                               batch_size=1024,
                               shuffle=False,
                               drop_last=False,
                               collate_fn=numpy_collate)
val_loader = data.DataLoader(val_set,
                            batch_size=1024,
                            shuffle=False,
                            drop_last=False,
                            collate_fn=numpy_collate)
test_loader = data.DataLoader(test_set,
                             batch_size=1024,
                             shuffle=False,
                             drop_last=False,
                             collate_fn=numpy_collate)

```

```

[12]: exmp_imgs = [train_set[i][0] for i in range(16)]
# Organize the images into a grid for nicer visualization
img_grid = torchvision.utils.make_grid(torch.from_numpy(np.stack(exmp_imgs, axis=0))[:, ,
←None],
                                       nrow=4,
                                       normalize=True,
                                       pad_value=0.5)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("FashionMNIST examples")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()

```

FashionMNIST examples



Visualizing the gradient flow after initialization

As mentioned previously, one important aspect of activation functions is how they propagate gradients through the network. Imagine we have a very deep neural network with more than 50 layers. The gradients for the input layer, i.e. the very first layer, have passed >50 times the activation function, but we still want them to be of a reasonable size. If the gradient through the activation function is (in expectation) considerably smaller than 1, our gradients will vanish until they reach the input layer. If the gradient through the activation function is larger than 1, the gradients exponentially increase and might explode.

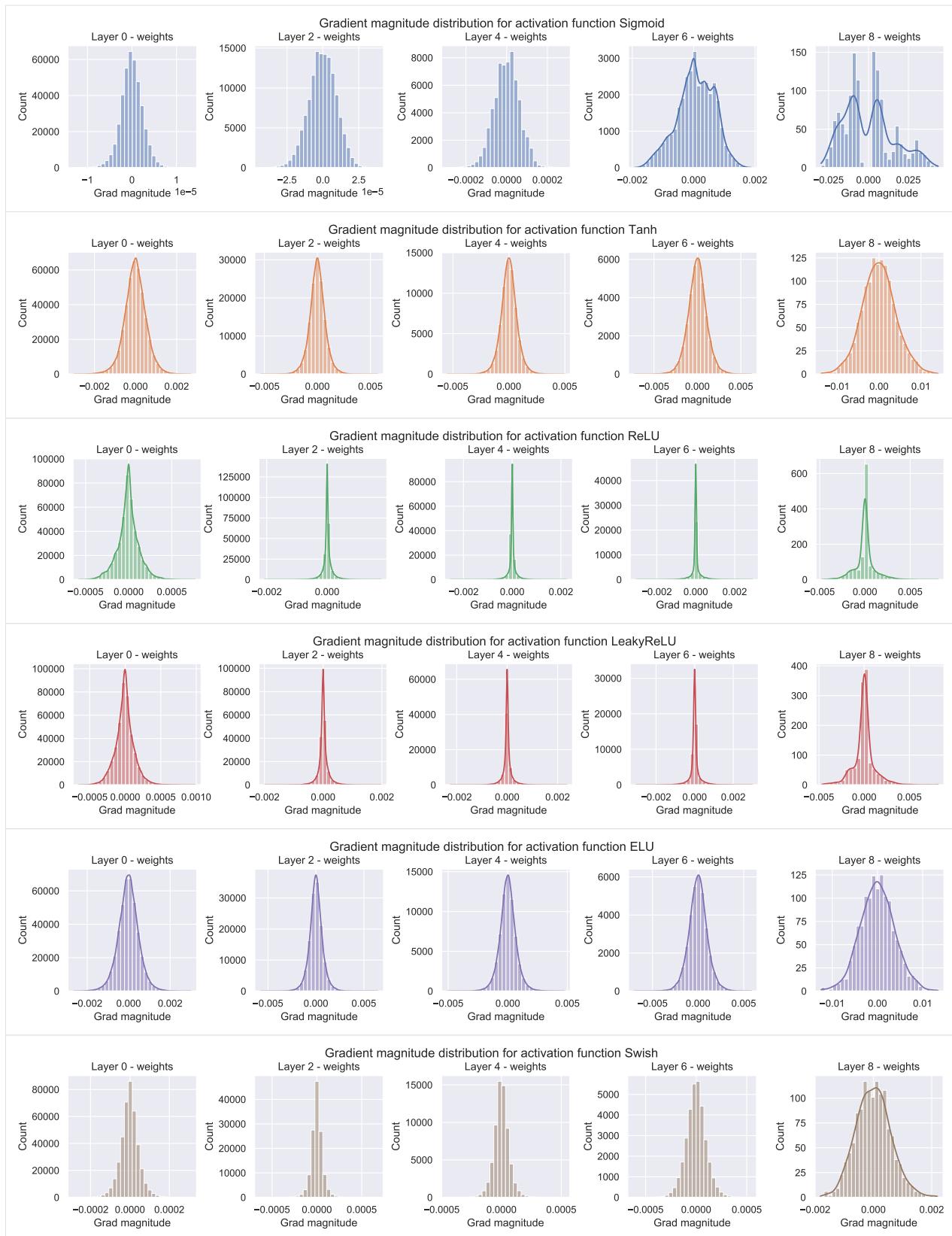
To get a feeling of how every activation function influences the gradients, we can look at a freshly initialized network and measure the gradients for each parameter for a batch of 256 images:

```
[13]: small_loader = data.DataLoader(train_set, batch_size=256, shuffle=False, collate_
    ↪fn=numpy_collate)
exmp_batch = next(iter(small_loader))

[14]: def visualize_gradients(net, params, color="C0"):
    """
    Inputs:
        net - Object of class BaseNetwork
        color - Color in which we want to visualize the histogram (for easier separation
    ↪of activation functions)
    """
    # Pass one batch through the network, and calculate the gradients for the weights
    def loss_func(p):
        imgs, labels = exmp_batch
        logits = net.apply(p, imgs)
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, labels).mean()
        return loss
    grads = jax.grad(loss_func)(params)
    grads = jax.device_get(grads)
    # We limit our visualization to the weight parameters and exclude the bias to reduce
    ↪the number of plots
    grads = jax.tree_util.tree_leaves(grads)
    grads = [g.reshape(-1) for g in grads if len(g.shape) > 1]

    ## Plotting
    columns = len(grads)
    fig, ax = plt.subplots(1, columns, figsize=(columns*3.5, 2.5))
    fig_index = 0
    for g_idx, g in enumerate(grads):
        key = f'Layer {g_idx * 2} - weights'
        key_ax = ax[g_idx%columns]
        sns.histplot(data=g, bins=30, ax=key_ax, color=color, kde=True)
        key_ax.set_title(str(key))
        key_ax.set_xlabel("Grad magnitude")
    fig.suptitle(f"Gradient magnitude distribution for activation function {net.act_fn.__
    ↪class__.__name__}", fontsize=14, y=1.05)
    fig.subplots_adjust(wspace=0.45)
    plt.show()
    plt.close()

[15]: # Seaborn prints warnings if histogram has small values. We can ignore them for now
import warnings
warnings.filterwarnings('ignore')
## Create a plot for every activation function
for i, act_fn_name in enumerate(act_fn_by_name):
    act_fn = act_fn_by_name[act_fn_name]()
    net_actfn = BaseNetwork(act_fn=act_fn)
    params = net_actfn.init(random.PRNGKey(0), exmp_batch[0])
    visualize_gradients(net_actfn, params, color=f'C{i}')
```



The sigmoid activation function shows a clearly undesirable behavior. While the gradients for the output layer are very

large with up to 0.1, the input layer has the lowest gradient norm across all activation functions with only 1e-5. This is due to its small maximum gradient of 1/4, and finding a suitable learning rate across all layers is not possible in this setup. All the other activation functions show to have similar gradient norms across all layers. Interestingly, the ReLU activation has a spike around 0 which is caused by its zero-part on the left, and dead neurons (we will take a closer look at this later on).

Note that additionally to the activation, the initialization of the weight parameters can be crucial. By default, PyTorch uses the [Kaiming](#) initialization for linear layers optimized for ReLU activations. In [Tutorial 4](#), we will take a closer look at initialization, but assume for now that the Kaiming initialization works for all activation functions reasonably well.

Training a model

Next, we want to train our model with different activation functions on FashionMNIST and compare the gained performance. All in all, our final goal is to achieve the best possible performance on a dataset of our choice. First, let's write the training and evaluation step that we can compile just-in-time (`jax.jit`) for efficiency:

```
[16]: def calculate_loss(params, apply_fn, batch):
    imgs, labels = batch
    logits = apply_fn(params, imgs)
    loss = optax.softmax_cross_entropy_with_integer_labels(logits, labels).mean()
    acc = (labels == logits.argmax(axis=-1)).mean()
    return loss, acc

@jax.jit
def train_step(state, batch):
    grad_fn = jax.value_and_grad(calculate_loss,
                                 has_aux=True)
    (_, acc), grads = grad_fn(state.params, state.apply_fn, batch)
    state = state.apply_gradients(grads=grads)
    return state, acc

@jax.jit
def eval_step(state, batch):
    _, acc = calculate_loss(state.params, state.apply_fn, batch)
    return acc
```

Using these functions, we write a training loop in the next cell including a validation after every epoch and a final test on the best model:

```
[17]: def train_model(net, model_name, max_epochs=50, patience=7, batch_size=256, ↴
    ↴overwrite=False):
    """
    Train a model on the training set of FashionMNIST

    Inputs:
        net - Object of BaseNetwork
        model_name - (str) Name of the model, used for creating the checkpoint names
        max_epochs - Number of epochs we want to (maximally) train for
        patience - If the performance on the validation set has not improved for
        ↴#patience epochs, we stop training early
        batch_size - Size of batches used in training
        overwrite - Determines how to handle the case when there already exists a
        ↴checkpoint. If True, it will be overwritten. Otherwise, we skip training.
```

(continues on next page)

(continued from previous page)

```

"""
file_exists = os.path.isfile(_get_model_file(CHECKPOINT_PATH, model_name))
if file_exists and not overwrite:
    print("Model file already exists. Skipping training...")
    state = None
else:
    if file_exists:
        print("Model file exists, but will be overwritten...")

    # Initializing parameters and training state
    params = net.init(random.PRNGKey(42), exmp_batch[0])
    state = train_state.TrainState.create(apply_fn=net.apply,
                                           params=params,
                                           tx=optax.sgd(learning_rate=1e-2,
                                                       momentum=0.9))

    # Defining data loader
    train_loader_local = data.DataLoader(train_set,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          drop_last=True,
                                          collate_fn= numpy_collate,
                                          generator=torch.Generator().manual_seed(42))

    val_scores = []
    best_val_epoch = -1
    for epoch in range(max_epochs):
        #####
        # Training #
        #####
        train_acc = 0.
        for batch in tqdm(train_loader_local, desc=f"Epoch {epoch+1}", leave=False):
            state, acc = train_step(state, batch)
            train_acc += acc
        train_acc /= len(train_loader_local)

        #####
        # Validation #
        #####
        val_acc = test_model(state, val_loader)
        val_scores.append(val_acc)
        print(f"[Epoch {epoch+1:2d}] Training accuracy: {train_acc:.05.2%}, "
              "Validation accuracy: {val_acc:.4%}")

        if len(val_scores) == 1 or val_acc > val_scores[best_val_epoch]:
            print("\t (New best performance, saving model...)")
            save_model(net, state.params, CHECKPOINT_PATH, model_name)
            best_val_epoch = epoch
        elif best_val_epoch <= epoch - patience:
            print(f"Early stopping due to no improvement over the last {patience} "
                  "epochs")
            break

```

(continues on next page)

(continued from previous page)

```

# Plot a curve of the validation accuracy
plt.plot([i for i in range(1,len(val_scores)+1)], val_scores)
plt.xlabel("Epochs")
plt.ylabel("Validation accuracy")
plt.title(f"Validation performance of {model_name}")
plt.show()
plt.close()

state, _ = load_model(CHECKPOINT_PATH, model_name, state=state)
test_acc = test_model(state, test_loader)
print(f" Test accuracy: {test_acc:4.2%} ").center(50, "=")+"\n"
return state, test_acc

def test_model(state, data_loader):
    """
    Test a model on a specified dataset.

    Inputs:
        state - Training state including parameters and model apply function.
        data_loader - DataLoader object of the dataset to test on (validation or test)
    """
    true_preds, count = 0., 0
    for batch in data_loader:
        acc = eval_step(state, batch)
        batch_size = batch[0].shape[0]
        true_preds += acc * batch_size
        count += batch_size
    test_acc = true_preds / count
    return test_acc

```

We train one model for each activation function. We recommend using the pretrained models to save time if you are running this notebook on CPU.

```
[18]: for act_fn_name in act_fn_by_name:
    print(f"Training BaseNetwork with {act_fn_name} activation...")
    act_fn = act_fn_by_name[act_fn_name]()
    net_actfn = BaseNetwork(act_fn=act_fn)
    train_model(net_actfn, f"FashionMNIST_{act_fn_name}", overwrite=False)
```

Training BaseNetwork with sigmoid activation...
 Model file already exists. Skipping training...
 ===== Test accuracy: 78.46% =====

Training BaseNetwork with tanh activation...
 Model file already exists. Skipping training...
 ===== Test accuracy: 88.57% =====

Training BaseNetwork with relu activation...
 Model file already exists. Skipping training...
 ===== Test accuracy: 88.65% =====

(continues on next page)

(continued from previous page)

```

Training BaseNetwork with leakyrelu activation...
Model file already exists. Skipping training...
===== Test accuracy: 88.33% =====

Training BaseNetwork with elu activation...
Model file already exists. Skipping training...
===== Test accuracy: 88.32% =====

Training BaseNetwork with swish activation...
Model file already exists. Skipping training...
===== Test accuracy: 88.33% =====

```

Surprisingly, in contrast to the [PyTorch implementation](#), the model using the sigmoid activation function actually trains. However, when looking at the validation curve, one can see that the model stays at random performance for many epochs, before it slowly starts learning. The different learning behavior can therefore be explained by small differences in, e.g., initialization or sampled batches.

All the other activation functions gain similar performance to each other and the [PyTorch implementation](#). To have a more accurate conclusion, we would have to train the models for multiple seeds and look at the averages. However, the “optimal” activation function also depends on many other factors (hidden sizes, number of layers, type of layers, task, dataset, optimizer, learning rate, initialization, etc.) so that a thorough grid search would not be useful in our case. In the literature, activation functions that have shown to work well with deep networks are all types of ReLU functions we experiment with here, with small gains for specific activation functions in specific networks.

Visualizing the activation distribution

After we have trained the models, we can look at the actual activation values that find inside the model. For instance, how many neurons are set to zero in ReLU? Where do we find most values in Tanh? To answer these questions, we can write a simple function which takes a trained model, applies it to a batch of images, and plots the histogram of the activations inside the network:

```
[19]: def visualize_activations(net, color="C0"):
    activations = []

    imgs, labels = exmp_batch
    _, activations = net(imgs, return_activations=True)

    ## Plotting
    columns = 4
    rows = math.ceil(len(activations)/columns)
    fig, ax = plt.subplots(rows, columns, figsize=(columns*2.7, rows*2.5))
    act_fn_name = net.act_fn.__class__.__name__
    for idx, activ in enumerate(activations):
        key_ax = ax[idx//columns][idx%columns]
        sns.histplot(data=activ.reshape(-1), bins=50, ax=key_ax, color=color, kde=True, stat="density")
        key_ax.set_title(f"Layer {idx} - {'Dense' if idx%2==0 else act_fn_name}")
    fig.suptitle("Activation distribution for activation function " + act_fn_name, fontsize=14)
    fig.subplots_adjust(hspace=0.4, wspace=0.4)
```

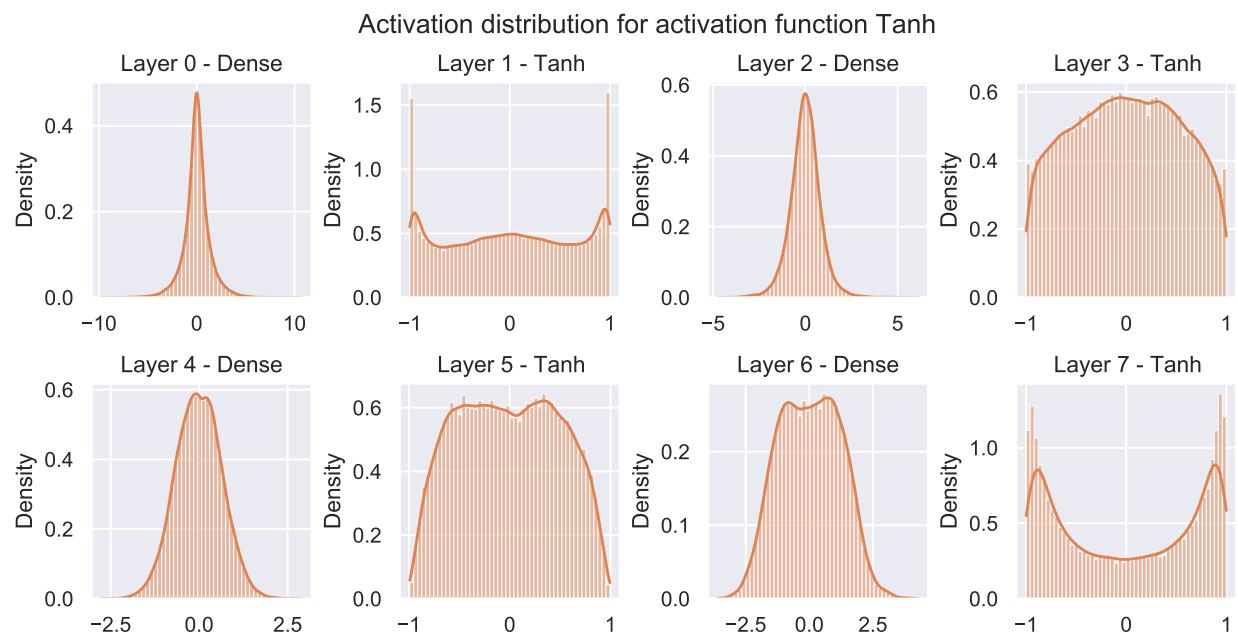
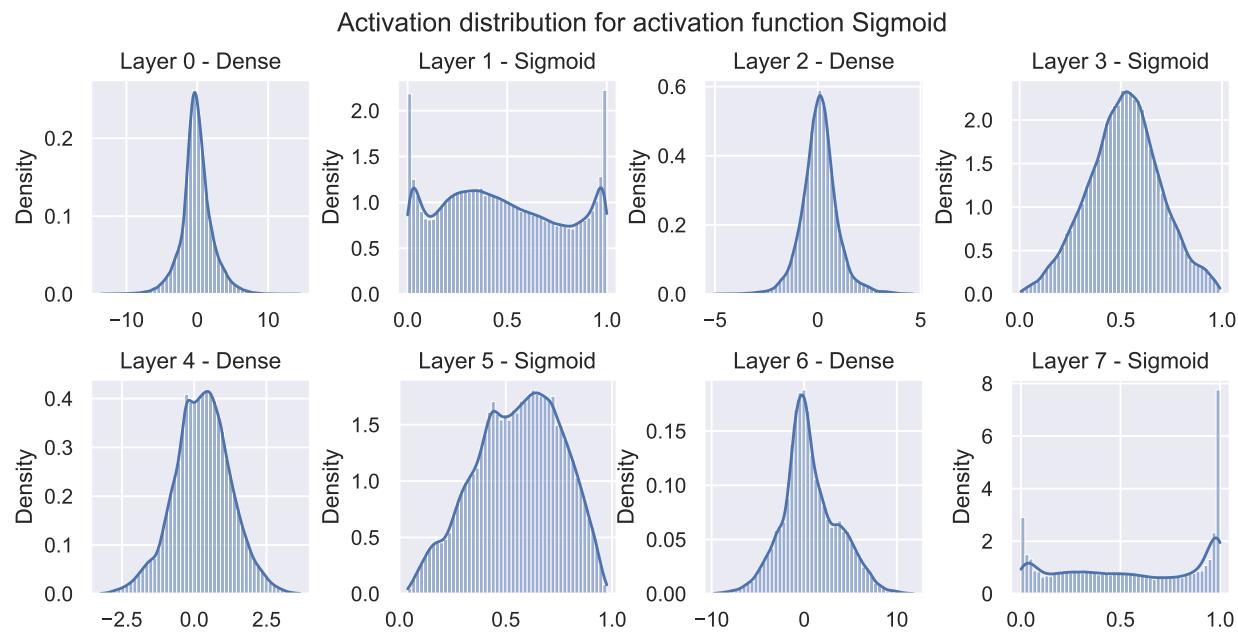
(continues on next page)

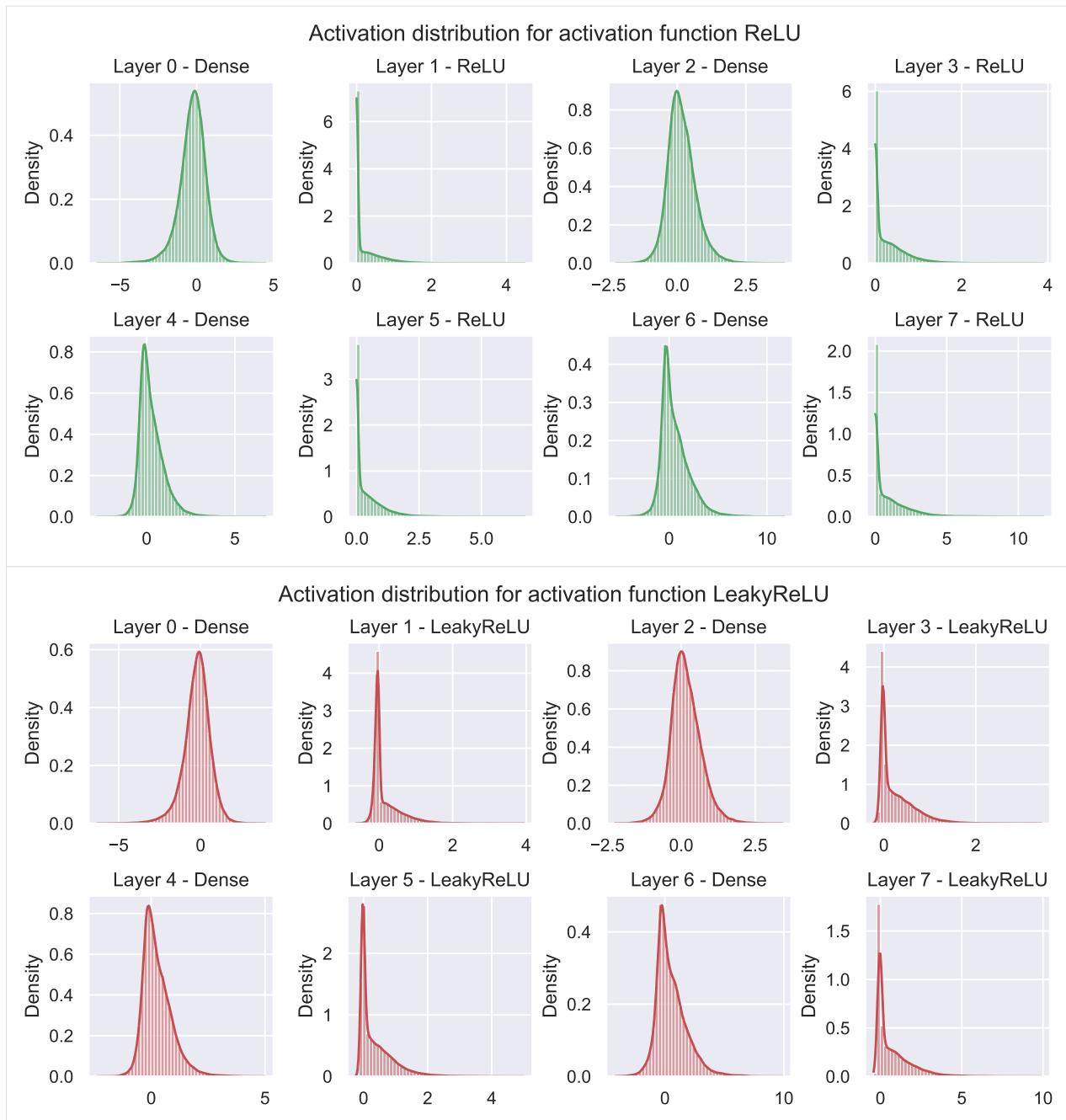
(continued from previous page)

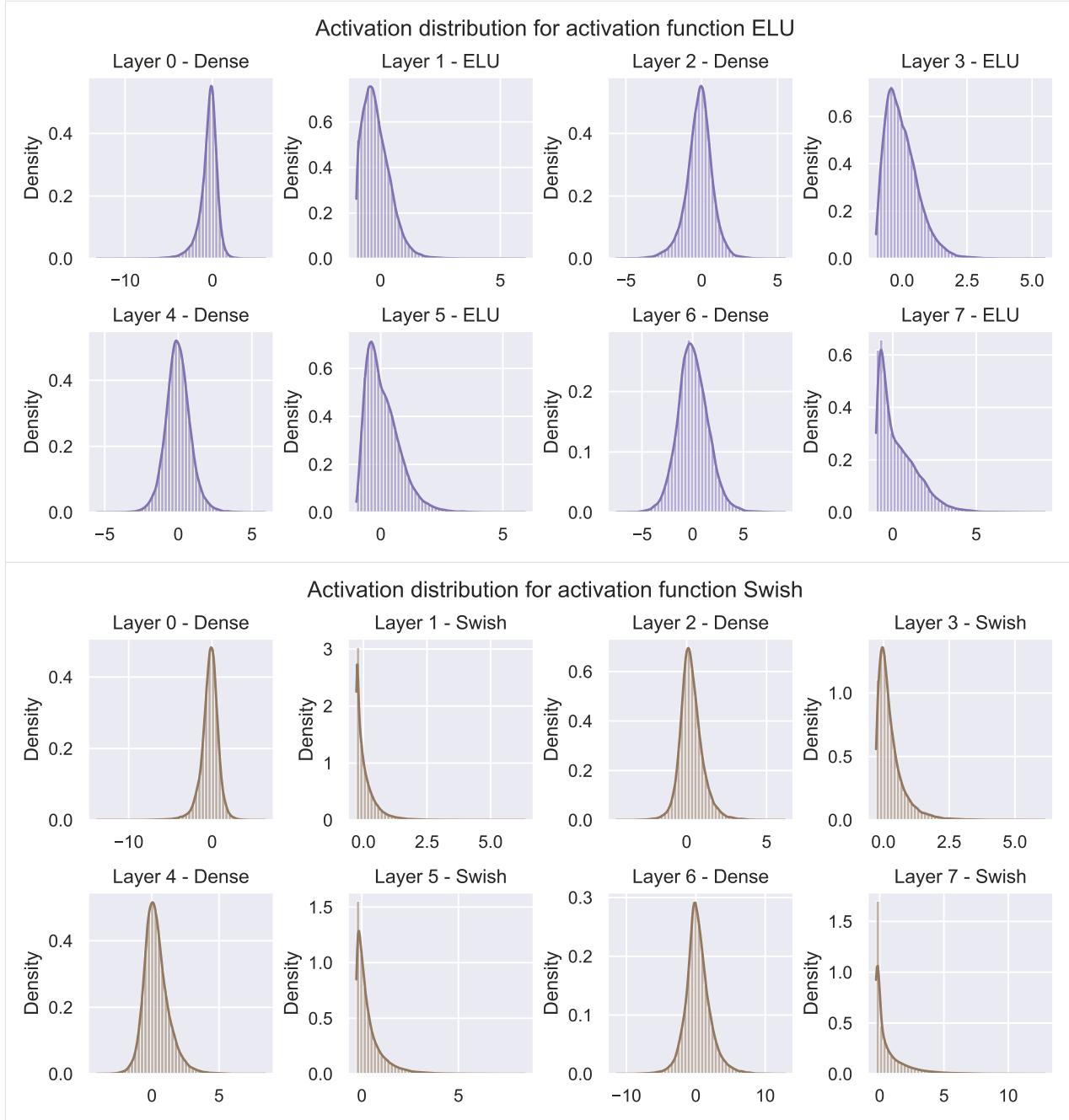
```
plt.show()
plt.close()
```

[20]:

```
for i, act_fn_name in enumerate(act_fn_by_name):
    state, net_actfn = load_model(model_path=CHECKPOINT_PATH, model_name=f"FashionMNIST_{act_fn_name}")
    net_actfn = net_actfn.bind(state.params)
    visualize_activations(net_actfn, color=f"C{i}")
```







For the model with sigmoid activation, we see a very high peak for the values 0 and 1 in the last layer. This suggests that the model is close to the saturation bounds, in which gradients do not properly flow back. Further, this is also the effect of the inconsistent gradient magnitudes we have seen before: since the layer closest to the output had the highest gradients, we see it is much stronger converged than, e.g., the second layer.

The tanh shows a more diverse behavior overall. While for the input layer we experience a larger amount of neurons to be close to -1 and 1, where the gradients are close to zero, the activations in the two consecutive layers are closer to zero. This is probably because the input layers look for specific features in the input image, and the consecutive layers combine those together. The activations for the last layer are again more biased to the extreme points because the classification layer can be seen as a weighted average of those values (the gradients push the activations to those extremes).

The ReLU has a strong peak at 0, as we initially expected. The effect of having no gradients for negative values is that the network does not have a Gaussian-like distribution after the linear layers, but a longer tail towards the positive values. The LeakyReLU shows a very similar behavior while ELU follows again a more Gaussian-like distribution. The Swish activation seems to lie in between, although it is worth noting that Swish uses significantly higher values than other activation functions (up to 15).

As all activation functions show slightly different behavior although obtaining similar performance for our simple network, it becomes apparent that the selection of the “optimal” activation function really depends on many factors, and is not the same for all possible networks.

Finding dead neurons in ReLU networks

One known drawback of the ReLU activation is the occurrence of “dead neurons”, i.e. neurons with no gradient for any training input. The issue of dead neurons is that as no gradient is provided for the layer, we cannot train the parameters of this neuron in the previous layer to obtain output values besides zero. For dead neurons to happen, the output value of a specific neuron of the linear layer before the ReLU has to be negative for all input images. Considering the large number of neurons we have in a neural network, it is not unlikely for this to happen.

To get a better understanding of how much of a problem this is, and when we need to be careful, we will measure how many dead neurons different networks have. For this, we implement a function which runs the network on the whole training set and records whether a neuron is exactly 0 for all data points or not:

```
[21]: def measure_number_dead_neurons(net):
    # For each neuron, we create a boolean variable initially set to 1. If it has an
    # activation unequal 0 at any time,
    # we set this variable to 0. After running through the whole training set, only dead
    # neurons will have a 1.
    neurons_dead = [
        jnp.ones(hd, dtype=jnp.dtype('bool')) for hd in net.hidden_sizes
    ] # Same shapes as hidden size in BaseNetwork

    get_activations = jax.jit(lambda inp: net(inp, return_activations=True)[1])
    for imgs, _ in tqdm(train_loader, leave=False): # Run through whole training set
        activations = get_activations(imgs)
        for layer_index, activ in enumerate(activations[1::2]):
            # Are all activations == 0 in the batch, and we did not record the opposite
            # in the last batches?
            neurons_dead[layer_index] = jnp.logical_and(neurons_dead[layer_index],_
            (activ == 0).all(axis=0))
    number_neurons_dead = [t.sum().item() for t in neurons_dead]
    print("Number of dead neurons:", number_neurons_dead)
    print("In percentage:", ", ".join([f"{num_dead / tens.shape[0]:.2%}" for tens, num_
    dead in zip(neurons_dead, number_neurons_dead)]))
```

First, we can measure the number of dead neurons for an untrained network:

```
[22]: net_relu = BaseNetwork(act_fn=ReLU())
params = net_relu.init(random.PRNGKey(42), exmp_batch[0])
measure_number_dead_neurons(net_relu.bind(params))

0% | 0/49 [00:00<?, ?it/s]

Number of dead neurons: [0, 0, 3, 7]
In percentage: 0.00%, 0.00%, 1.17%, 5.47%
```

We see that only a minor amount of neurons are dead, but that they increase with the depth of the layer. However, this is not a problem for the small number of dead neurons we have as the input to later layers is changed due to updates to the weights of previous layers. Therefore, dead neurons in later layers can potentially become “alive”/active again.

How does this look like for a trained network (with the same initialization)?

```
[23]: state, net_relu = load_model(model_path=CHECKPOINT_PATH, model_name="FashionMNIST_relu")
measure_number_dead_neurons(net_relu.bind(state.params))

0%|          | 0/49 [00:00<?, ?it/s]

Number of dead neurons: [0, 0, 3, 3]
In percentage: 0.00%, 0.00%, 1.17%, 2.34%
```

The number of dead neurons indeed decreased in the later layers. However, it should be noted that dead neurons are especially problematic in the input layer. As the input does not change over epochs (the training set is kept as it is), training the network cannot turn those neurons back active. Still, the input data has usually a sufficiently high standard deviation to reduce the risk of dead neurons.

Finally, we check how the number of dead neurons behaves with increasing layer depth. For instance, let's take the following 10-layer neural network:

```
[24]: net_relu = BaseNetwork(act_fn=ReLU(), hidden_sizes=[256, 256, 256, 256, 256, 256, 128, 128, 128, 128])
params = net_relu.init(random.PRNGKey(42), exmp_batch[0])
measure_number_dead_neurons(net_relu.bind(params))

0%|          | 0/49 [00:00<?, ?it/s]

Number of dead neurons: [0, 1, 1, 19, 75, 53, 71, 57, 64, 69]
In percentage: 0.00%, 0.39%, 0.39%, 7.42%, 29.30%, 41.41%, 55.47%, 44.53%, 50.00%, 53.91%
```

The number of dead neurons is significantly higher than before which harms the gradient flow especially in the first iterations. For instance, more than 56% of the neurons in the pre-last layer are dead which creates a considerable bottleneck. Hence, it is advisable to use other nonlinearities like Swish for very deep networks.

4.20.3 Conclusion

In this notebook, we have reviewed a set of six activation functions (sigmoid, tanh, ReLU, LeakyReLU, ELU, and Swish) in neural networks, and discussed how they influence the gradient distribution across layers. Sigmoid tends to fail in deep neural networks as the highest gradient it provides is 0.25 leading to vanishing gradients in early layers. All ReLU-based activation functions have shown to perform well, and besides the original ReLU, do not have the issue of dead neurons. When implementing your own neural network, it is recommended to start with a ReLU-based network and select the specific activation function based on the properties of the network.

4.20.4 References

- [1] Ramachandran, Prajit, Barret Zoph, and Quoc V. Le. “Searching for activation functions.” arXiv preprint arXiv:1710.05941 (2017). [Paper link](#)
-

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.21 Tutorial 4 (JAX): Optimization and Initialization

Filled notebook:

Pre-trained models:

PyTorch version:

Author: Phillip Lippe

Note: This notebook is written in JAX+Flax. It is a 1-to-1 translation of the original notebook written in PyTorch+PyTorch Lightning with almost identical results. For an introduction to JAX, check out our [Tutorial 2 \(JAX\): Introduction to JAX+Flax](#). Further, throughout the notebook, we comment on major differences to the PyTorch version and provide explanations for the major parts of the JAX code. We do not provide speed comparisons for this notebook since they tend to be uninformative for such small networks and potentially bottlenecked by other factors.

In this tutorial, we will review techniques for optimization and initialization of neural networks. When increasing the depth of neural networks, there are various challenges we face. Most importantly, we need to have a stable gradient flow through the network, as otherwise, we might encounter vanishing or exploding gradients. This is why we will take a closer look at the following concepts: initialization and optimization.

In the first half of the notebook, we will review different initialization techniques, and go step by step from the simplest initialization to methods that are nowadays used in very deep networks. In the second half, we focus on optimization comparing the optimizers SGD, SGD with Momentum, and Adam.

Let's start with importing our standard libraries:

```
[1]: ## Standard libraries
import os
import json
import math
import numpy as np
import copy
from typing import Any, Sequence, Callable, NamedTuple, Optional, Tuple
PyTree = Any # Type definition for PyTree, for readability
from copy import deepcopy
import pickle

## Imports for plotting
import matplotlib.pyplot as plt
from matplotlib import cm
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
import seaborn as sns
sns.set()
```

(continues on next page)

(continued from previous page)

```

## Progress bar
from tqdm.auto import tqdm

## JAX
import jax
import jax.numpy as jnp
from jax import random
from jax.tree_util import tree_map

## Flax (NN in JAX)
try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax
from flax import linen as nn
from flax.training import train_state, checkpoints

## Optax (Optimizers in JAX)
try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax

/home/phillip/anaconda3/envs/dl2020/lib/python3.7/site-packages/chex/_src/pytypes.py:37: FutureWarning: jax.tree_structure is deprecated, and will be removed in a future release. Use jax.tree_util.tree_structure instead.
PyTreeDef = type(jax.tree_structure(None))
WARNING:absl:GlobalAsyncCheckpointManager is not imported correctly. Checkpointing of GlobalDeviceArrays will not be available.To use the feature, install tensorstore.

```

We will use the same path variables DATASET_PATH and CHECKPOINT_PATH as in Tutorial 3. Adjust the paths if necessary.

```
[2]: # Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../../saved_models/tutorial4_jax"

# Verifying the device that will be used throughout this notebook
print("Device:", jax.devices()[0])

Device: gpu:0
```

In the last part of the notebook, we will train models using three different optimizers. The pretrained models for those are downloaded below.

```
[3]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/JAX/tutorial4/"
# Files to download
```

(continues on next page)

(continued from previous page)

```

pretrained_files = ["FashionMNIST_SGD.config",      "FashionMNIST_SGD_results.json",
                    "FashionMNIST_SGD.tar",           "FashionMNIST_SGDMom.config", "FashionMNIST_SGDMom_results.json",
                    "FashionMNIST_SGDMom.tar",        "FashionMNIST_Adam.config",   "FashionMNIST_Adam_results.json",
                    "FashionMNIST_Adam.tar"       ]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive",
                  "folder, or contact the author with the full output including the following error:\n", e)

```

4.21.1 Preparation

Throughout this notebook, we will use a deep fully connected network, similar to our previous tutorial. We will also again apply the network to FashionMNIST, so you can relate to the results of Tutorial 3. We start by loading the FashionMNIST dataset:

```

[4]: import torch
import torch.utils.data as data
from torchvision.datasets import FashionMNIST
from torchvision import transforms

# Transformations applied on each image => bring them into a numpy array and normalize,
# to mean 0 and std 1
def image_to_numpy(img):
    img = np.array(img, dtype=np.float32)
    img = (img / 255. - 0.2861) / 0.3530
    return img

# We need to stack the batch elements as numpy arrays
def numpy_collate(batch):
    if isinstance(batch[0], np.ndarray):
        return np.stack(batch)
    elif isinstance(batch[0], (tuple, list)):
        transposed = zip(*batch)
        return [numpy_collate(samples) for samples in transposed]
    else:
        return np.array(batch)

```

(continues on next page)

(continued from previous page)

```

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = FashionMNIST(root=DATASET_PATH,
                             train=True,
                             transform=image_to_numpy,
                             download=True)
train_set, val_set = torch.utils.data.random_split(train_dataset,
                                                   [50000, 10000],
                                                   generator=torch.Generator().manual_
→seed(42))

# Loading the test set
test_set = FashionMNIST(root=DATASET_PATH,
                         train=False,
                         transform=image_to_numpy,
                         download=True)

# We define a set of data loaders that we can use for various purposes later.
# Note that for actually training a model, we will use different data loaders
# with a lower batch size.
train_loader = data.DataLoader(train_set,
                               batch_size=1024,
                               shuffle=False,
                               drop_last=False,
                               collate_fn=numpy_collate)
val_loader = data.DataLoader(val_set,
                            batch_size=1024,
                            shuffle=False,
                            drop_last=False,
                            collate_fn=numpy_collate)
test_loader = data.DataLoader(test_set,
                             batch_size=1024,
                             shuffle=False,
                             drop_last=False,
                             collate_fn=numpy_collate)

```

In comparison to the previous tutorial, we have changed the parameters of the normalization transformation in `image_to_numpy`. The normalization is now designed to give us an expected mean of 0 and a standard deviation of 1 across pixels. This will be particularly relevant for the discussion about initialization we will look at below, and hence we change it here. It should be noted that in most classification tasks, both normalization techniques (between -1 and 1 or mean 0 and stddev 1) have shown to work well. We can calculate the normalization parameters by determining the mean and standard deviation on the original images:

```
[5]: print("Mean", (train_dataset.data.float() / 255.0).mean().item())
print("Std", (train_dataset.data.float() / 255.0).std().item())

Mean 0.28604060411453247
Std 0.35302424443084717
```

We can verify the transformation by looking at the statistics of a single batch:

```
[6]: imgs, _ = next(iter(train_loader))
print(f"Mean: {imgs.mean().item():.3f}")
print(f"Standard deviation: {imgs.std().item():.3f}")
```

(continues on next page)

(continued from previous page)

```
print(f"Maximum: {imgs.max().item():.3f}")
print(f"Minimum: {imgs.min().item():.3f}")

Mean: 0.008
Standard deviation: 1.009
Maximum: 2.022
Minimum: -0.810
```

Note that the maximum and minimum are not 1 and -1 anymore, but shifted towards the positive values. This is because FashionMNIST contains a lot of black pixels, similar to MNIST.

Next, we create a linear neural network. We use the same setup as in the previous tutorial.

```
[7]: # Network
class BaseNetwork(nn.Module):
    act_fn : Callable
    num_classes : int = 10
    hidden_sizes : Sequence = (512, 256, 256, 128)
    kernel_init : Callable = nn.linear.default_kernel_init

    @nn.compact
    def __call__(self, x, return_activations=False):
        x = x.reshape(x.shape[0], -1) # Reshape images to a flat vector
        # We collect all activations throughout the network for later visualizations
        # Remember that in jitted functions, unused tensors will anyways be removed.
        activations = []
        for hd in self.hidden_sizes:
            x = nn.Dense(hd,
                         kernel_init=self.kernel_init)(x)
            activations.append(x)
            x = self.act_fn(x)
            activations.append(x)
        x = nn.Dense(self.num_classes,
                     kernel_init=self.kernel_init)(x)
        activations.append(x)
        return x if not return_activations else (x, activations)
```

For the activation functions, we make use of JAX's and Flax's library instead of implementing ourselves. However, we also define an `Identity` activation function. Although this activation function would significantly limit the network's modeling capabilities, we will use it in the first steps of our discussion about initialization (for simplicity).

```
[8]: act_fn_by_name = {
    "tanh": nn.tanh,
    "relu": nn.relu,
    "identity": lambda x: x
}
```

Finally, we define a few plotting functions that we will use for our discussions. These functions help us to (1) visualize the weight/parameter distribution inside a network, (2) visualize the gradients that the parameters at different layers receive, and (3) the activations, i.e. the output of the linear layers. The detailed code is not important, but feel free to take a closer look if interested.

```
[9]: #####
```

(continues on next page)

(continued from previous page)

```

def plot_dists(val_dict, color="C0", xlabel=None, stat="count", use_kde=True):
    columns = len(val_dict)
    fig, ax = plt.subplots(1, columns, figsize=(columns*3, 2.5))
    fig_index = 0
    for key in sorted(val_dict.keys()):
        key_ax = ax[fig_index%columns]
        sns.histplot(val_dict[key], ax=key_ax, color=color, bins=50, stat=stat,
                     kde=use_kde and ((val_dict[key].max()-val_dict[key].min())>1e-8)) #_
        ↵Only plot kde if there is variance
        key_ax.set_title(f'{key} ' + (r'(%i $\to$ %i)' % (val_dict[key].shape[1], val_
        ↵dict[key].shape[0])) if len(val_dict[key].shape)>1 else ""))
        if xlabel is not None:
            key_ax.set_xlabel(xlabel)
        fig_index += 1
    fig.subplots_adjust(wspace=0.4)
    return fig

#####
def visualize_weight_distribution(params, color="C0"):
    params, _ = jax.tree_util.tree_flatten(params)
    params = [p.reshape(-1) for p in params if len(p.shape) > 1] # Remove biases
    params = jax.device_get(params)
    weights = {f'Layer {layer_idx*2}': p for layer_idx, p in enumerate(params)}

    ## Plotting
    fig = plot_dists(weights, color=color, xlabel="Weight vals")
    fig.suptitle("Weight distribution", fontsize=14, y=1.05)
    plt.show()
    plt.close()

#####
small_loader = data.DataLoader(train_set, batch_size=1024, shuffle=False, collate_
    ↵fn=numpy_collate)
exmp_imgs, exmp_labels = next(iter(small_loader))

def visualize_gradients(model, params, color="C0", print_variance=False):
    """
    Inputs:
        net - Object of class BaseNetwork
        color - Color in which we want to visualize the histogram (for easier separation_
        ↵of activation functions)
    """
    # Pass one batch through the network, and calculate the gradients for the weights
    def loss_func(p):
        logits = model.apply(p, exmp_imgs)
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, exmp_labels).
        ↵mean()
        return loss
    grads = jax.grad(loss_func)(params)
    grads = jax.device_get(grads)

```

(continues on next page)

(continued from previous page)

```

# We limit our visualization to the weight parameters and exclude the bias to reduce
# the number of plots
grads = jax.tree_util.tree_leaves(grads)
grads = [g.reshape(-1) for g in grads if len(g.shape) > 1]
grads = {f'Layer {layer_idx*2}': g for layer_idx, g in enumerate(grads)}

## Plotting
fig = plot_dists(grads, color=color, xlabel="Grad magnitude")
fig.suptitle("Gradient distribution", fontsize=14, y=1.05)
plt.show()
plt.close()

if print_variance:
    for key in sorted(grads.keys()):
        print(f"{key} - Variance: {np.var(grads[key])}")

#####
def visualize_activations(model, params, color="C0", print_variance=False):
    # Pass one batch through the network, and calculate the activations
    _, activations = model.apply(params, exmp_imgs, return_activations=True)
    activations = {f'Layer {layer_idx*2}': act.reshape(-1) for layer_idx, act in
    enumerate(activations[::2])}

    ## Plotting
    fig = plot_dists(activations, color=color, stat="density", xlabel="Activation vals")
    fig.suptitle("Activation distribution", fontsize=14, y=1.05)
    plt.show()
    plt.close()

    if print_variance:
        for key in sorted(activations.keys()):
            print(f"{key} - Variance: {np.var(activations[key])}")

#####

```

4.21.2 Initialization

Before starting our discussion about initialization, it should be noted that there exist many very good blog posts about the topic of neural network initialization (for example [deeplearning.ai](#), or a more [math-focused blog post](#)). In case something remains unclear after this tutorial, we recommend skimming through these blog posts as well.

When initializing a neural network, there are a few properties we would like to have. First, the variance of the input should be propagated through the model to the last layer, so that we have a similar standard deviation for the output neurons. If the variance would vanish the deeper we go in our model, it becomes much harder to optimize the model as the input to the next layer is basically a single constant value. Similarly, if the variance increases, it is likely to explode (i.e. head to infinity) the deeper we design our model. The second property we look out for in initialization techniques is a gradient distribution with equal variance across layers. If the first layer receives much smaller gradients than the last layer, we will have difficulties in choosing an appropriate learning rate.

As a starting point for finding a good method, we will analyze different initialization based on our linear neural network

with no activation function (i.e. an identity). We do this because initializations depend on the specific activation function used in the network, and we can adjust the initialization schemes later on for our specific choice.

```
[10]: def init_simple_model(kernel_init, act_fn=act_fn_by_name['identity']):
    model = BaseNetwork(act_fn=act_fn,
                        kernel_init=kernel_init)
    params = model.init(random.PRNGKey(42), exmp_imgs)
    return model, params
```

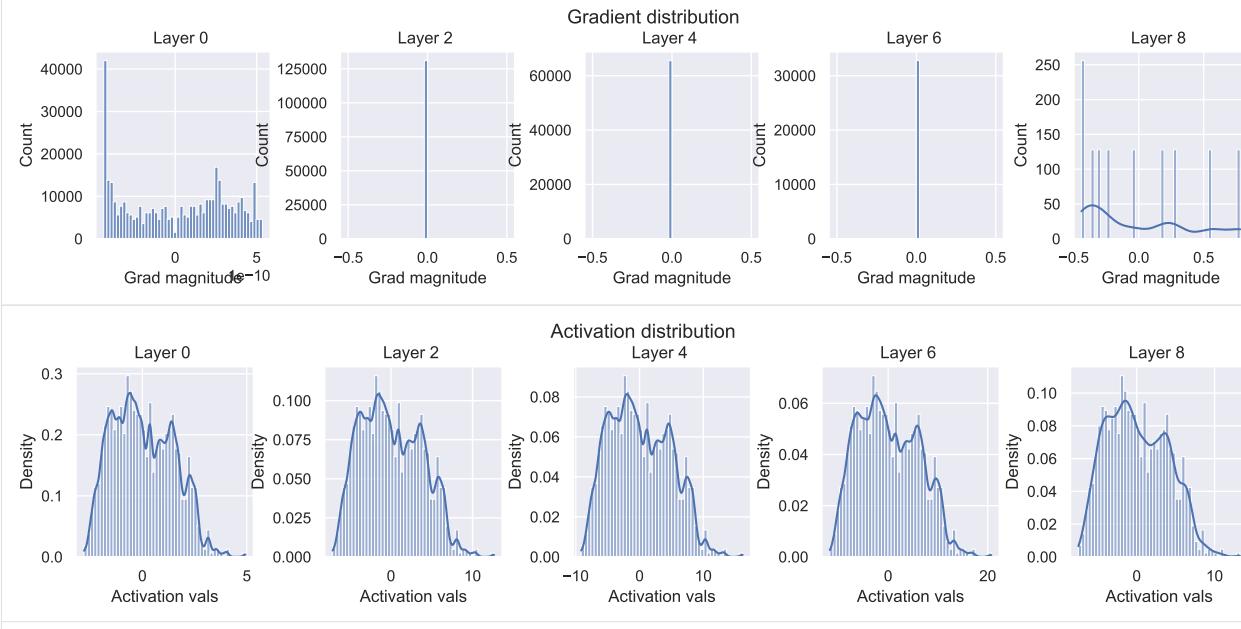
Note that in contrast to the PyTorch version of this tutorial, we keep the bias initialization fixed. As the default, JAX uses a zero initialization. In practice, the initialization of the bias is less relevant as long as it is reasonably close to zero, since it is only a (learnable) constant added to the features.

Constant initialization

The first initialization we can consider is to initialize all weights with the same constant value. Intuitively, setting all weights to zero is not a good idea as the propagated gradient will be zero. However, what happens if we set all weights to a value slightly larger or smaller than 0? To find out, we can implement a function for setting all parameters below and visualize the gradients.

```
[11]: # An initialization function in JAX takes as input a PRNG key,
# the shape of the parameter to create, and the data type
# (usually jnp.float32). We create this function based on the
# input parameter 'c' here, indicating the constant value
def get_const_init_func(c=0.0):
    return lambda key, shape, dtype: c*jnp.ones(shape, dtype=dtype)

model, params = init_simple_model(get_const_init_func(c=0.005))
visualize_gradients(model, params)
visualize_activations(model, params, print_variance=True)
```



Layer 0 - Variance: 2.0582759380340576

Layer 2 - Variance: 13.489117622375488

Layer 4 - Variance: 22.100563049316406

(continues on next page)

(continued from previous page)

```
Layer 6 - Variance: 36.20956039428711
Layer 8 - Variance: 14.831436157226562
```

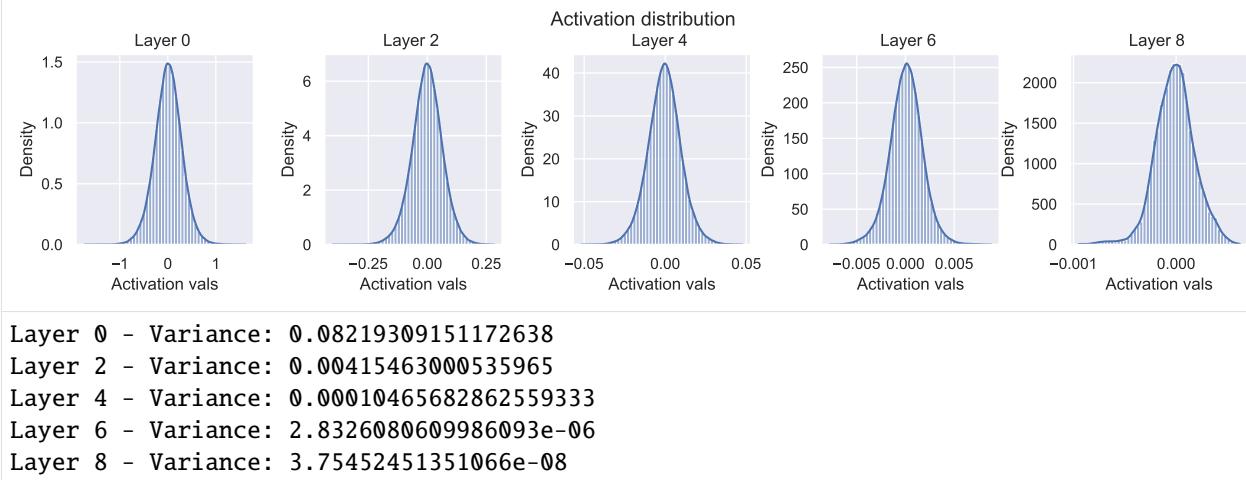
As we can see, only the first and the last layer have diverse gradient distributions while the other three layers have the same gradient for all weights (note that this value is unequal 0, but often very close to it). Having the same gradient for parameters that have been initialized with the same values means that we will always have the same value for those parameters. This would make our layer useless and reduce our effective number of parameters to 1. Thus, we cannot use a constant initialization to train our networks.

Constant variance

From the experiment above, we have seen that a constant value is not working. So instead, how about we initialize the parameters by randomly sampling from a distribution like a Gaussian? The most intuitive way would be to choose one variance that is used for all layers in the network. Let's implement it below, and visualize the activation distribution across layers.

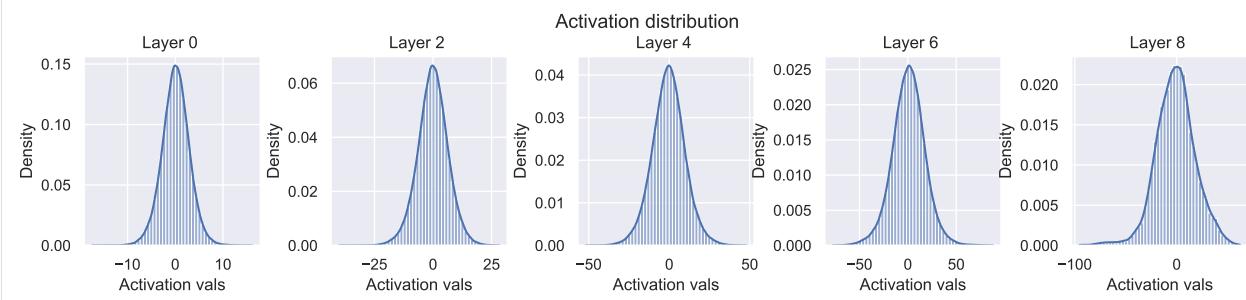
```
[12]: def get_var_init_func(std=0.01):
    return lambda key, shape, dtype: std*random.normal(key, shape, dtype=dtype)

model, params = init_simple_model(get_var_init_func(std=0.01))
visualize_activations(model, params, print_variance=True)
```



The variance of the activation becomes smaller and smaller across layers, and almost vanishes in the last layer. Alternatively, we could use a higher standard deviation:

```
[13]: model, params = init_simple_model(get_var_init_func(std=0.1))
visualize_activations(model, params, print_variance=True)
```



```

Layer 0 - Variance: 8.21930980682373
Layer 2 - Variance: 41.54630661010742
Layer 4 - Variance: 104.6568603515625
Layer 6 - Variance: 283.26092529296875
Layer 8 - Variance: 375.4525451660156

```

With a higher standard deviation, the activations are likely to explode. You can play around with the specific standard deviation values, but it will be hard to find one that gives us a good activation distribution across layers and is very specific to our model. If we would change the hidden sizes or number of layers, you would have to search all over again, which is neither efficient nor recommended.

How to find appropriate initialization values

From our experiments above, we have seen that we need to sample the weights from a distribution, but are not sure which one exactly. As a next step, we will try to find the optimal initialization from the perspective of the activation distribution. For this, we state two requirements:

1. The mean of the activations should be zero
2. The variance of the activations should stay the same across every layer

Suppose we want to design an initialization for the following layer: $y = Wx + b$ with $y \in \mathbb{R}^{d_y}$, $x \in \mathbb{R}^{d_x}$. Our goal is that the variance of each element of y is the same as the input, i.e. $\text{Var}(y_i) = \text{Var}(x_i) = \sigma_x^2$, and that the mean is zero. We assume x to also have a mean of zero, because, in deep neural networks, y would be the input of another layer. This requires the bias and weight to have an expectation of 0. Actually, as b is a single element per output neuron and is constant across different inputs, we set it to 0 overall.

Next, we need to calculate the variance with which we need to initialize the weight parameters. Along the calculation, we will need the following variance rule: given two independent variables, the variance of their product is $\text{Var}(X \cdot Y) = \mathbb{E}(Y^2)\text{Var}(X) + \mathbb{E}(X^2)\text{Var}(Y) + \text{Var}(X)\text{Var}(Y) = \mathbb{E}(Y^2)\mathbb{E}(X^2) - \mathbb{E}(Y)^2\mathbb{E}(X)^2$ (X and Y are not referring to x and y , but any random variable).

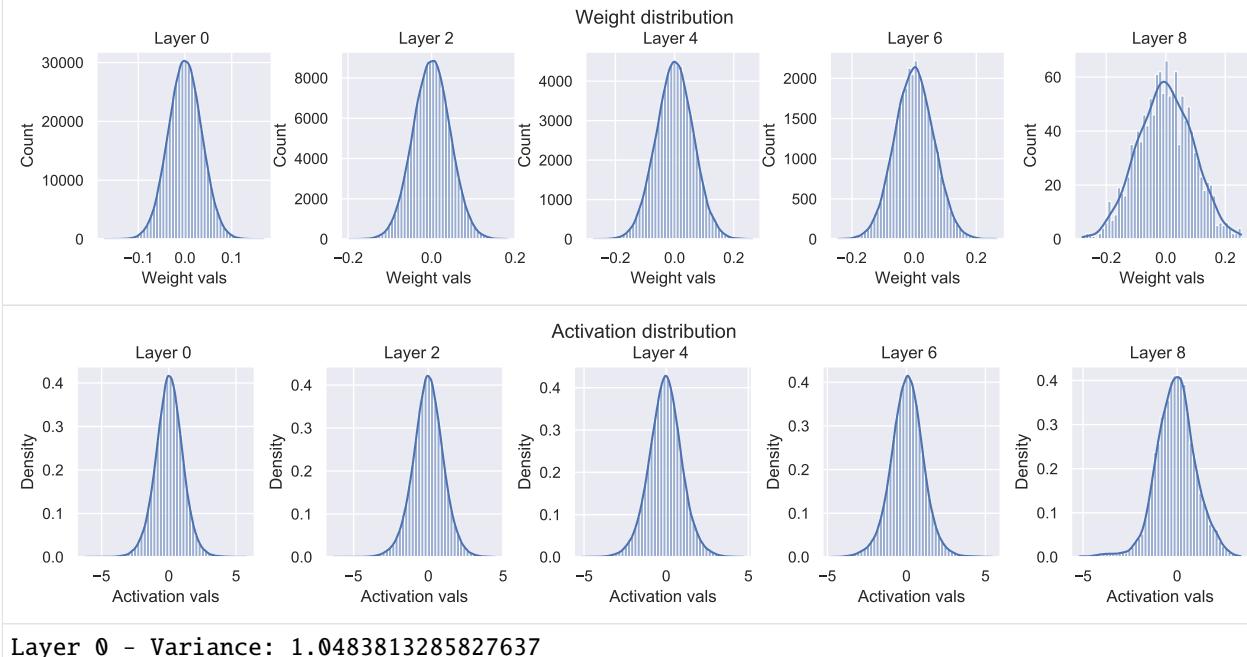
The needed variance of the weights, $\text{Var}(w_{ij})$, is calculated as follows:

$$\begin{aligned}
y_i &= \sum_j w_{ij} x_j && \text{Calculation of a single output neuron without bias} \\
\text{Var}(y_i) &= \sigma_x^2 = \text{Var}\left(\sum_j w_{ij} x_j\right) \\
&= \sum_j \text{Var}(w_{ij} x_j) && \text{Inputs and weights are independent of each other} \\
&= \sum_j \text{Var}(w_{ij}) \cdot \text{Var}(x_j) && \text{Variance rule (see above) with expectations being zero} \\
&= d_x \cdot \text{Var}(w_{ij}) \cdot \text{Var}(x_j) && \text{Variance equal for all } d_x \text{ elements} \\
&= \sigma_x^2 \cdot d_x \cdot \text{Var}(w_{ij}) \\
\Rightarrow \text{Var}(w_{ij}) &= \sigma_W^2 = \frac{1}{d_x}
\end{aligned}$$

Thus, we should initialize the weight distribution with a variance of the inverse of the input dimension d_x . Let's implement it below and check whether this holds:

```
[14]: equal_var_init = lambda key, shape, dtype: 1.0/np.sqrt(shape[0]) * random.normal(key, shape, dtype=dtype)
```

```
model, params = init_simple_model(equal_var_init)
visualize_weight_distribution(params)
visualize_activations(model, params, print_variance=True)
```



```
Layer 0 - Variance: 1.0483813285827637
Layer 2 - Variance: 1.0350143909454346
Layer 4 - Variance: 1.018454670906067
Layer 6 - Variance: 1.0767643451690674
Layer 8 - Variance: 1.1150107383728027
```

As we expected, the variance stays indeed constant across layers. Note that our initialization does not restrict us to a normal distribution, but allows any other distribution with a mean of 0 and variance of $1/d_x$. You often see that a uniform distribution is used for initialization. A small benefit of using a uniform instead of a normal distribution is that we can exclude the chance of initializing very large or small weights.

Besides the variance of the activations, another variance we would like to stabilize is the one of the gradients. This ensures a stable optimization for deep networks. It turns out that we can do the same calculation as above starting from $\Delta x = W \Delta y$, and come to the conclusion that we should initialize our layers with $1/d_y$ where d_y is the number of output neurons. You can do the calculation as a practice, or check a thorough explanation in [this blog post](#). As a compromise between both constraints, [Glorot and Bengio \(2010\)](#) proposed to use the harmonic mean of both values. This leads us to the well-known Xavier initialization:

$$W \sim \mathcal{N} \left(0, \frac{2}{d_x + d_y} \right)$$

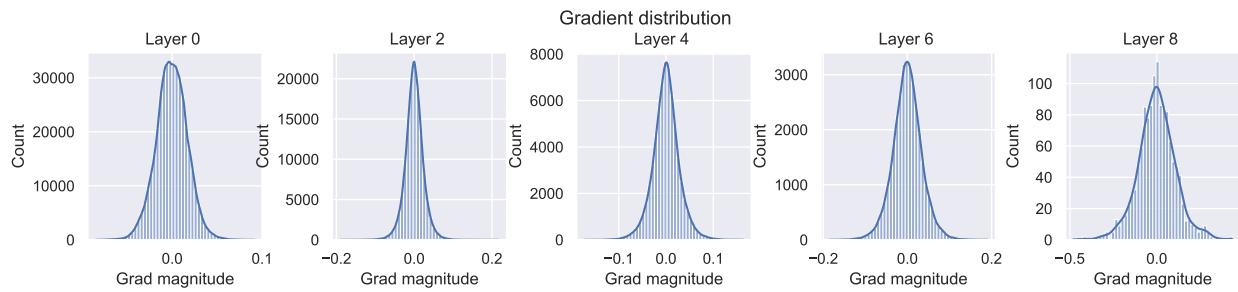
If we use a uniform distribution, we would initialize the weights with:

$$W \sim U \left[-\frac{\sqrt{6}}{\sqrt{d_x + d_y}}, \frac{\sqrt{6}}{\sqrt{d_x + d_y}} \right]$$

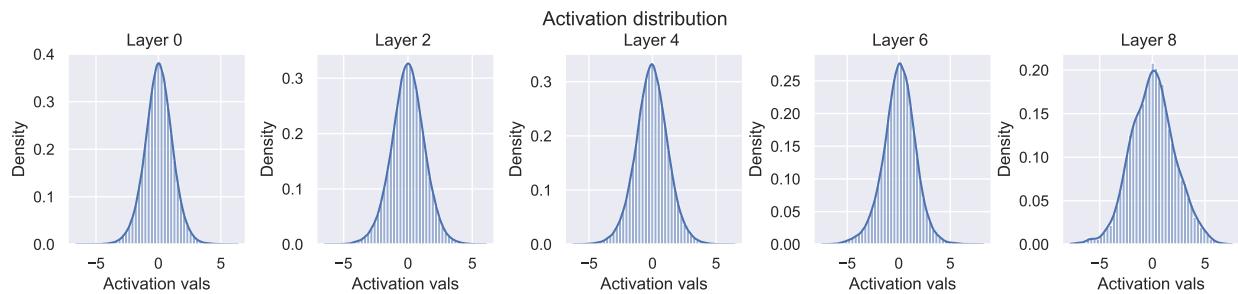
Let's shortly implement it and validate its effectiveness:

```
[15]: def xavier_init(key, shape, dtype):
    bound = math.sqrt(6)/math.sqrt(shape[0]+shape[1])
    return random.uniform(key, shape, dtype,
                          minval=-bound, maxval=bound)

model, params = init_simple_model(xavier_init)
visualize_gradients(model, params, print_variance=True)
visualize_activations(model, params, print_variance=True)
```



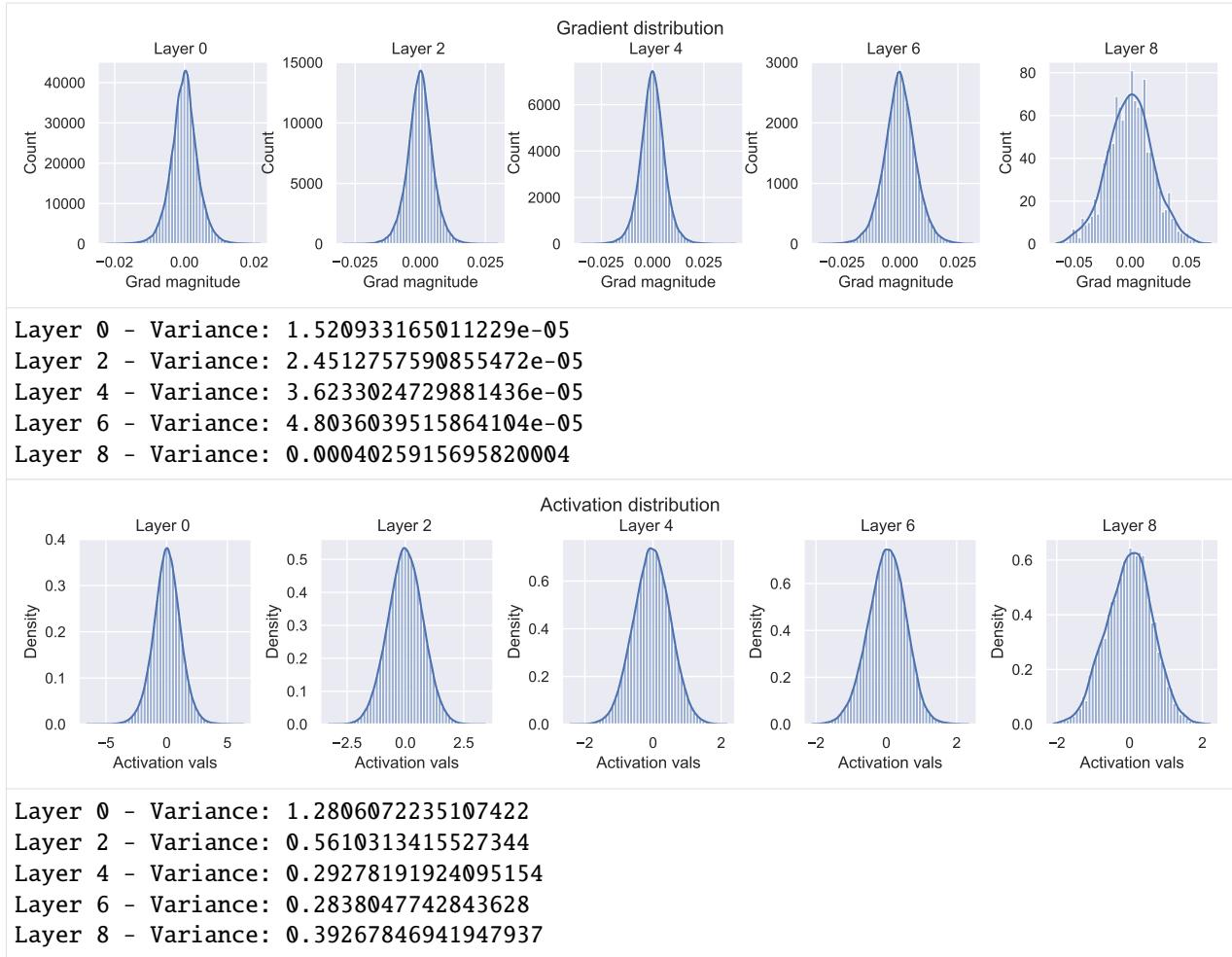
```
Layer 0 - Variance: 0.00029831365100108087
Layer 2 - Variance: 0.0005463268025778234
Layer 4 - Variance: 0.0007796576828695834
Layer 6 - Variance: 0.0011752902064472437
Layer 8 - Variance: 0.011736484244465828
```



```
Layer 0 - Variance: 1.2806072235107422
Layer 2 - Variance: 1.6784099340438843
Layer 4 - Variance: 1.6767947673797607
Layer 6 - Variance: 2.5439999103546143
Layer 8 - Variance: 4.41333532333374
```

We see that the Xavier initialization balances the variance of gradients and activations. Note that the significantly higher variance for the output layer is due to the large difference of input and output dimension (128 vs 10). However, we currently assumed the activation function to be linear. So what happens if we add a non-linearity? In a tanh-based network, a common assumption is that for small values during the initial steps in training, the tanh works as a linear function such that we don't have to adjust our calculation. We can check if that is the case for us as well:

```
[16]: model, params = init_simple_model(xavier_init, act_fn=nn.tanh)
visualize_gradients(model, params, print_variance=True)
visualize_activations(model, params, print_variance=True)
```



Although the variance decreases over depth, it is apparent that the activation distribution becomes more focused on the low values. Therefore, our variance will stabilize around 0.25 if we would go even deeper. Hence, we can conclude that the Xavier initialization works well for Tanh networks. But what about ReLU networks? Here, we cannot take the previous assumption of the non-linearity becoming linear for small values. The ReLU activation function sets (in expectation) half of the inputs to 0 so that also the expectation of the input is not zero. However, as long as the expectation of W is zero and $b = 0$, the expectation of the output is zero. The part where the calculation of the ReLU initialization differs from the identity is when determining $\text{Var}(w_{ij}x_j)$:

$$\text{Var}(w_{ij}x_j) = \underbrace{\mathbb{E}[w_{ij}^2]}_{=\text{Var}(w_{ij})} \mathbb{E}[x_j^2] - \underbrace{\mathbb{E}[w_{ij}]^2}_{=0} \mathbb{E}[x_j]^2 = \text{Var}(w_{ij})\mathbb{E}[x_j^2]$$

If we assume now that x is the output of a ReLU activation (from a previous layer, $x = \max(0, \tilde{y})$), we can calculate the expectation as follows:

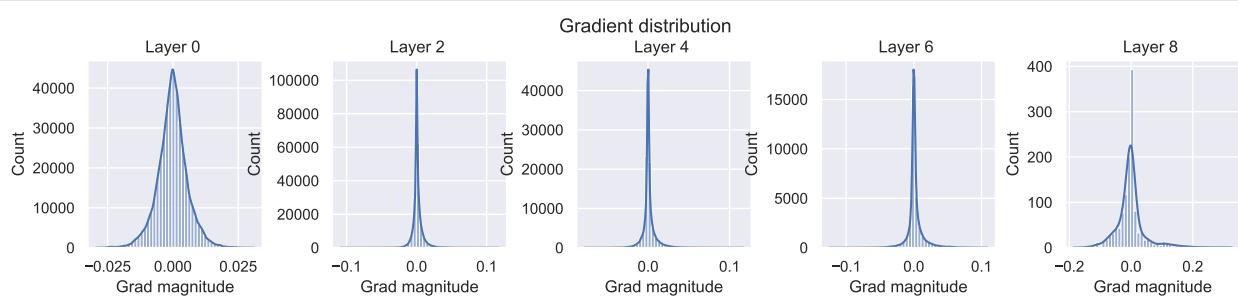
$$\begin{aligned}\mathbb{E}[x^2] &= \mathbb{E}[\max(0, \tilde{y})^2] \\ &= \frac{1}{2}\mathbb{E}[\tilde{y}^2] \quad \tilde{y} \text{ is zero-centered and symmetric} \\ &= \frac{1}{2}\text{Var}(\tilde{y})\end{aligned}$$

Thus, we see that we have an additional factor of 1/2 in the equation, so that our desired weight variance becomes $2/d_x$. This gives us the Kaiming initialization (see He, K. et al. (2015)). Note that the Kaiming initialization does not use the harmonic mean between input and output size. In their paper (Section 2.2, Backward Propagation, last paragraph),

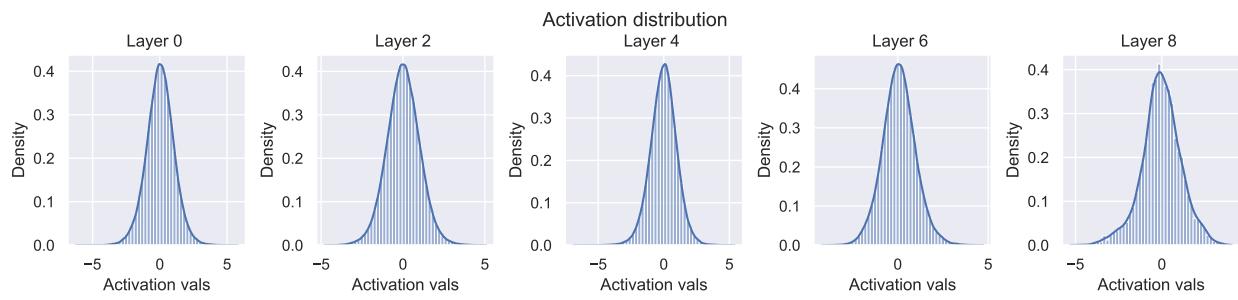
they argue that using d_x or d_y both lead to stable gradients throughout the network, and only depend on the overall input and output size of the network. Hence, we can use here only the input d_x :

```
[17]: num_input_feats = np.prod(exmp_imgs.shape[1:])
def kaiming_init(key, shape, dtype):
    # The first layer does not have ReLU applied on its input
    # Note that this detection only works if we do not use 784
    # feature size anywhere - better to explicitly handle
    # layer numbers
    if shape[0] == num_input_feats:
        std = 1/np.sqrt(shape[0])
    else:
        std = np.sqrt(2/shape[0])
    return std * random.normal(key, shape, dtype)

model, params = init_simple_model(kaiming_init, act_fn=nn.relu)
visualize_gradients(model, params, print_variance=True)
visualize_activations(model, params, print_variance=True)
```



Layer 0 - Variance: 3.25944201904349e-05
 Layer 2 - Variance: 4.664550942834467e-05
 Layer 4 - Variance: 6.605686212424189e-05
 Layer 6 - Variance: 0.00015451121726073325
 Layer 8 - Variance: 0.002561226487159729



Layer 0 - Variance: 1.0483813285827637
 Layer 2 - Variance: 1.0137324333190918
 Layer 4 - Variance: 0.9987665414810181
 Layer 6 - Variance: 0.8383973836898804
 Layer 8 - Variance: 1.316560983657837

The variance stays stable across layers. We can conclude that the Kaiming initialization indeed works well for ReLU-based networks. Note that for Leaky-ReLU etc., we have to slightly adjust the factor of 2 in the variance as half of the values are not set to zero anymore.

4.21.3 Optimization

Besides initialization, selecting a suitable optimization algorithm can be an important choice for deep neural networks. Before taking a closer look at them, we should define code for training the models. Most of the following code is copied from the previous tutorial, and only slightly altered to fit our needs.

```
[18]: def _get_config_file(model_path, model_name):
    # Name of the file for storing hyperparameter details
    return os.path.join(model_path, model_name + ".config")

def _get_model_file(model_path, model_name):
    # Name of the file for storing network parameters
    return os.path.join(model_path, model_name + ".tar")

def _get_result_file(model_path, model_name):
    return os.path.join(model_path, model_name + "_results.json")

def load_model(model_path, model_name, state=None):
    """
    Loads a saved model from disk.

    Inputs:
        model_path - Path of the checkpoint directory
        model_name - Name of the model (str)
        state - (Optional) If given, the parameters are loaded into this training state.
    ↪Otherwise,
                a new one is created alongside a network architecture.
    """
    config_file, model_file = _get_config_file(model_path, model_name), _get_model_file(model_path, model_name)
    assert os.path.isfile(config_file), f"Could not find the config file \'{config_file}\'. Are you sure this is the correct path and you have your model config stored here?"
    assert os.path.isfile(model_file), f"Could not find the model file \'{model_file}\'. Are you sure this is the correct path and you have your model stored here?"
    with open(config_file, "r") as f:
        config_dict = json.load(f)
    if state is None:
        net = BaseNetwork(act_fn=nn.relu, **config_dict)
        state = train_state.TrainState(step=0,
                                        params=None,
                                        apply_fn=net.apply,
                                        tx=None,
                                        opt_state=None)
    else:
        net = None
    # You can also use flax's checkpoint package. To show an alternative,
    # you can instead load the parameters simply from a pickle file.
    with open(model_file, 'rb') as f:
        params = pickle.load(f)
    state = state.replace(params=params)
    return state, net

def save_model(model, params, model_path, model_name):
```

(continues on next page)

(continued from previous page)

```

"""
Given a model, we save the parameters and hyperparameters.

Inputs:
    model - Network object without parameters
    params - Parameters to save of the model
    model_path - Path of the checkpoint directory
    model_name - Name of the model (str)
"""

config_dict = {'hidden_sizes': model.hidden_sizes,
               'num_classes': model.num_classes}
os.makedirs(model_path, exist_ok=True)
config_file, model_file = _get_config_file(model_path, model_name), _get_model_
file(model_path, model_name)
with open(config_file, "w") as f:
    json.dump(config_dict, f)
# You can also use flax's checkpoint package. To show an alternative,
# you can instead save the parameters simply in a pickle file.
with open(model_file, 'wb') as f:
    pickle.dump(params, f)

def calculate_loss(params, apply_fn, batch):
    imgs, labels = batch
    logits = apply_fn(params, imgs)
    loss = optax.softmax_cross_entropy_with_integer_labels(logits, labels).mean()
    acc = (labels == logits.argmax(axis=-1)).mean()
    return loss, acc

@jax.jit
def train_step(state, batch):
    grad_fn = jax.value_and_grad(calculate_loss,
                                 has_aux=True)
    (_, acc), grads = grad_fn(state.params, state.apply_fn, batch)
    state = state.apply_gradients(grads=grads)
    return state, acc

@jax.jit
def eval_step(state, batch):
    _, acc = calculate_loss(state.params, state.apply_fn, batch)
    return acc

def train_model(net, params, optimizer, model_name, max_epochs=50, batch_size=256,_
overwrite=False):
    """
    Train a model on the training set of FashionMNIST

    Inputs:
        net - Object of BaseNetwork
        params - The parameters to use as initialization
        optimizer - Optimizer to use
        model_name - (str) Name of the model, used for creating the checkpoint names
        max_epochs - Number of epochs we want to (maximally) train for
    """

```

(continues on next page)

(continued from previous page)

```

batch_size - Size of batches used in training
overwrite - Determines how to handle the case when there already exists a
↪checkpoint. If True, it will be overwritten. Otherwise, we skip training.
"""

file_exists = os.path.isfile(_get_model_file(CHECKPOINT_PATH, model_name))
if file_exists and not overwrite:
    print("Model file already exists. Skipping training...")
    state = None
    with open(_get_result_file(CHECKPOINT_PATH, model_name), "r") as f:
        results = json.load(f)
else:
    if file_exists:
        print("Model file exists, but will be overwritten...")

    # Initializing training state
    results = None
    state = train_state.TrainState.create(apply_fn=net.apply,
                                           params=params,
                                           tx=optimizer)

    # Defining data loader
    train_loader_local = data.DataLoader(train_set,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          drop_last=True,
                                          collate_fn= numpy_collate,
                                          generator=torch.Generator().manual_seed(42))

    train_scores = []
    val_scores = []
    best_val_epoch = -1
    for epoch in range(max_epochs):
        #####
        # Training #
        #####
        train_acc = 0.
        for batch in tqdm(train_loader_local, desc=f"Epoch {epoch+1}", leave=False):
            state, acc = train_step(state, batch)
            train_acc += acc
        train_acc /= len(train_loader_local)
        train_scores.append(train_acc.item())

        #####
        # Validation #
        #####
        val_acc = test_model(state, val_loader)
        val_scores.append(val_acc)
        print(f"[Epoch {epoch+1}] Training accuracy: {train_acc:.2%}, ↪
↪Validation accuracy: {val_acc:.2%}")

        if len(val_scores) == 1 or val_acc > val_scores[best_val_epoch]:
            print("\t (New best performance, saving model...)")

```

(continues on next page)

(continued from previous page)

```

        save_model(net, state.params, CHECKPOINT_PATH, model_name)
        best_val_epoch = epoch

    state, _ = load_model(CHECKPOINT_PATH, model_name, state=state)
    if results is None:
        test_acc = test_model(state, test_loader)
        results = {"test_acc": test_acc, "val_scores": val_scores,
                   "train_scores": train_scores}
    with open(_get_result_file(CHECKPOINT_PATH, model_name), "w") as f:
        json.dump(results, f)

    # Plot a curve of the validation accuracy
    sns.set()
    plt.plot([i for i in range(1, len(results["train_scores"])+1)], results["train_scores"],
             label="Train")
    plt.plot([i for i in range(1, len(results["val_scores"])+1)], results["val_scores"],
             label="Val")
    plt.xlabel("Epochs")
    plt.ylabel("Validation accuracy")
    plt.ylim(min(results["val_scores"]), max(results["train_scores"])*1.01)
    plt.title(f"Validation performance of {model_name}")
    plt.legend()
    plt.show()
    plt.close()

    print((f" Test accuracy: {results['test_acc']:.2%} ").center(50, "=")+"\n")
    return state

def test_model(state, data_loader):
    """
    Test a model on a specified dataset.

    Inputs:
        state - Training state including parameters and model apply function.
        data_loader - DataLoader object of the dataset to test on (validation or test)
    """
    true_preds, count = 0., 0
    for batch in data_loader:
        acc = eval_step(state, batch)
        batch_size = batch[0].shape[0]
        true_preds += acc * batch_size
        count += batch_size
    test_acc = true_preds / count
    return test_acc.item()

```

First, we need to understand what an optimizer actually does. The optimizer is responsible to update the network's parameters given the gradients. Hence, we effectively implement a function $w^t = f(w^{t-1}, g^t, \dots)$ with w being the parameters, and $g^t = \nabla_{w^{(t-1)}} \mathcal{L}^{(t)}$ the gradients at time step t . A common, additional parameter to this function is the learning rate, here denoted by η . Usually, the learning rate can be seen as the “step size” of the update. A higher learning rate means that we change the weights more in the direction of the gradients, a smaller means we take shorter steps.

As most optimizers only differ in the implementation of f , we can define a template for an optimizer in JAX below. Note that in comparison to PyTorch, the optimizers must be functional and have no effects outside the function. Hence, we follow the basic implementation principles of `optax` and define an optimizer to be a tuple of two main functions:

- `init`: Given a set of parameters, initialize any state variables that the optimizer may need. While it might not be necessary for the simplest optimizers, more sophisticated like Adam need to keep track of gradient statistics, which can be initialized here.
- `update`: Given the gradients, the optimizer state, and the parameters, return the updates that should be applied to the parameters, and an updated optimizer state.

The updates can be applied to the parameters using `optax.apply_updates(params, updates)` which adds each update value to its respective parameter. In simplified form, it applies `tree_map(lambda p, u: p + u, params, updates)`. As a reminder, `jax.tree_util.tree_map` is a function that runs over PyTrees and applies a function to all leafs, returning a new PyTree with the results. If we input multiple PyTrees, it runs over them synchronously and applies the functions on tuples of leafs, one of each PyTree. In the function above, this means that a new PyTree is created for which each leaf has the sum of the respective leaf in `params` and `updates`.

In contrast to PyTorch, there is no `zero_grad` function since the gradients are the output of a separate gradient function, not an object-based backpropagation through a dynamic computation graph. The template is setup below:

```
[19]: class Optimizer(NamedTuple):
    # Given the parameters, initialize any optimizer state as tuple
    init : Callable[[PyTree], tuple]
    # Given the gradients, optimizer state, and evt. parameters, return
    # the parameter updates and new optimizer state
    update : Callable[[PyTree, tuple, Optional[PyTree]], Tuple[PyTree, tuple]]
```

The first optimizer we are going to implement is the standard Stochastic Gradient Descent (SGD). SGD updates the parameters using the following equation:

$$w^{(t)} = w^{(t-1)} - \eta \cdot g^{(t)}$$

As simple as the equation is also our implementation of SGD:

```
[20]: def sgd(lr):
    def init(params):
        return tuple()

    def update(updates, state, params=None):
        updates = tree_map(lambda u: -lr * u, updates)
        return updates, state

    return Optimizer(init, update)
```

Note that we define SGD as a function that returns a tuple of the `init` and `update` function, which depends on the input parameter `lr`, i.e., the learning rate. SGD does not have any state it needs to keep track of. Hence, the `init` function returns an empty tuple. The `update` function simply multiplies all gradients with the learning rate. The negative sign is needed since the `apply_updates` function *adds* the updates to the parameters, not subtracts.

In the lecture, we also have discussed the concept of momentum which replaces the gradient in the update by an exponential average of all past gradients including the current one:

$$\begin{aligned} m^{(t)} &= \beta_1 m^{(t-1)} + (1 - \beta_1) \cdot g^{(t)} \\ w^{(t)} &= w^{(t-1)} - \eta \cdot m^{(t)} \end{aligned}$$

Let's also implement it below:

```
[21]: def sgd_momentum(lr, momentum=0.0):
    def init(params):
        param_momentum = tree_map(jnp.zeros_like, params)
        return param_momentum

    def update(updates, state, params=None):
        state = tree_map(lambda m, g: (1 - momentum) * g + momentum * m,
                        state,
                        updates)
        updates = tree_map(lambda m: - lr * m,
                           state)
        return updates, state

    return Optimizer(init, update)
```

We track the momentum parameters $m^{(t)}$ in a PyTree, which is initialized with all zeros, but with the same shape and structure as the parameters. At each update iteration, we update our momentum parameters with above's equation, and use it to calculate the new updates.

As a third and final optimizer, we arrive at Adam. Adam combines the idea of momentum with an adaptive learning rate, which is based on an exponential average of the squared gradients, i.e. the gradients norm. Furthermore, we add a bias correction for the momentum and adaptive learning rate for the first iterations:

$$\begin{aligned} m^{(t)} &= \beta_1 m^{(t-1)} + (1 - \beta_1) \cdot g^{(t)} \\ v^{(t)} &= \beta_2 v^{(t-1)} + (1 - \beta_2) \cdot (g^{(t)})^2 \\ \hat{m}^{(t)} &= \frac{m^{(t)}}{1 - \beta_1^t}, \hat{v}^{(t)} = \frac{v^{(t)}}{1 - \beta_2^t} \\ w^{(t)} &= w^{(t-1)} - \frac{\eta}{\sqrt{\hat{v}^{(t)}} + \epsilon} \circ \hat{m}^{(t)} \end{aligned}$$

Epsilon is a small constant used to improve numerical stability for very small gradient norms. Remember that the adaptive learning rate does not replace the learning rate hyperparameter η , but rather acts as an extra factor and ensures that the gradients of various parameters have a similar norm.

```
[22]: def adam(lr, beta1=0.9, beta2=0.999, eps=1e-8):
    def init(params):
        step = 0.
        param_momentum = tree_map(jnp.zeros_like, params)
        param_2nd_momentum = tree_map(jnp.zeros_like, params)
        return (step, param_momentum, param_2nd_momentum)

    def update(updates, state, params=None):
        # Update momentum and adapt. lr
        step, param_momentum, param_2nd_momentum = state
        step += 1
        param_momentum = tree_map(lambda m, g: (1 - beta1) * g + beta1 * m,
                                  param_momentum,
                                  updates)
        param_2nd_momentum = tree_map(lambda m2, g: (1 - beta2) * g**2 + beta2 * m2,
                                      param_2nd_momentum,
                                      updates)
        # Calculate update for single parameter
        def update_param(m, m2):
            pass
```

(continues on next page)

(continued from previous page)

```

# Bias correction
m /= 1 - beta1 ** step
m2 /= 1 - beta2 ** step
return - m * lr / (jnp.sqrt(m2) + eps)
# Update for all parameters
updates = tree_map(update_param,
                    param_momentum,
                    param_2nd_momentum)
return updates, (step, param_momentum, param_2nd_momentum)

return Optimizer(init, update)

```

Our state consists of a counter `step` that keeps track of the time step t , the momentums $m^{(t)}$ and the second momentum/adaptive learning rate parameters $v^{(t)}$. At each update step, we update $m^{(t)}$ and $v^{(t)}$, and afterwards calculate the updates for each parameters based on $m^{(t)}$ and $v^{(t)}$.

Comparing optimizers on model training

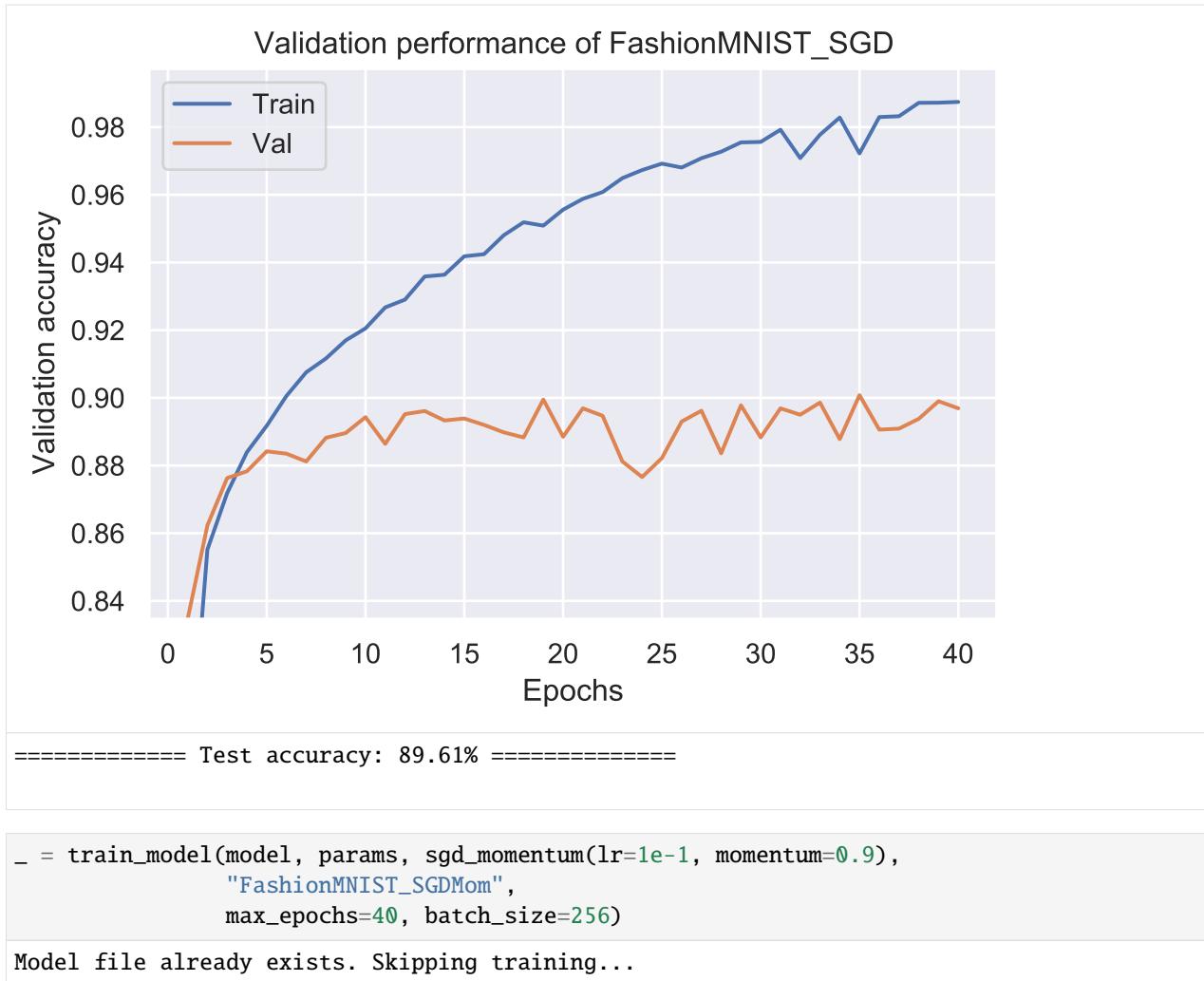
After we have implemented three optimizers (SGD, SGD with momentum, and Adam), we can start to analyze and compare them. First, we test them on how well they can optimize a neural network on the FashionMNIST dataset. We use again our linear network, this time with a ReLU activation and the kaiming initialization, which we have found before to work well for ReLU-based networks. Note that the model is over-parameterized for this task, and we can achieve similar performance with a much smaller network (for example 100, 100, 100). However, our main interest is in how well the optimizer can train *deep* neural networks, hence the over-parameterization.

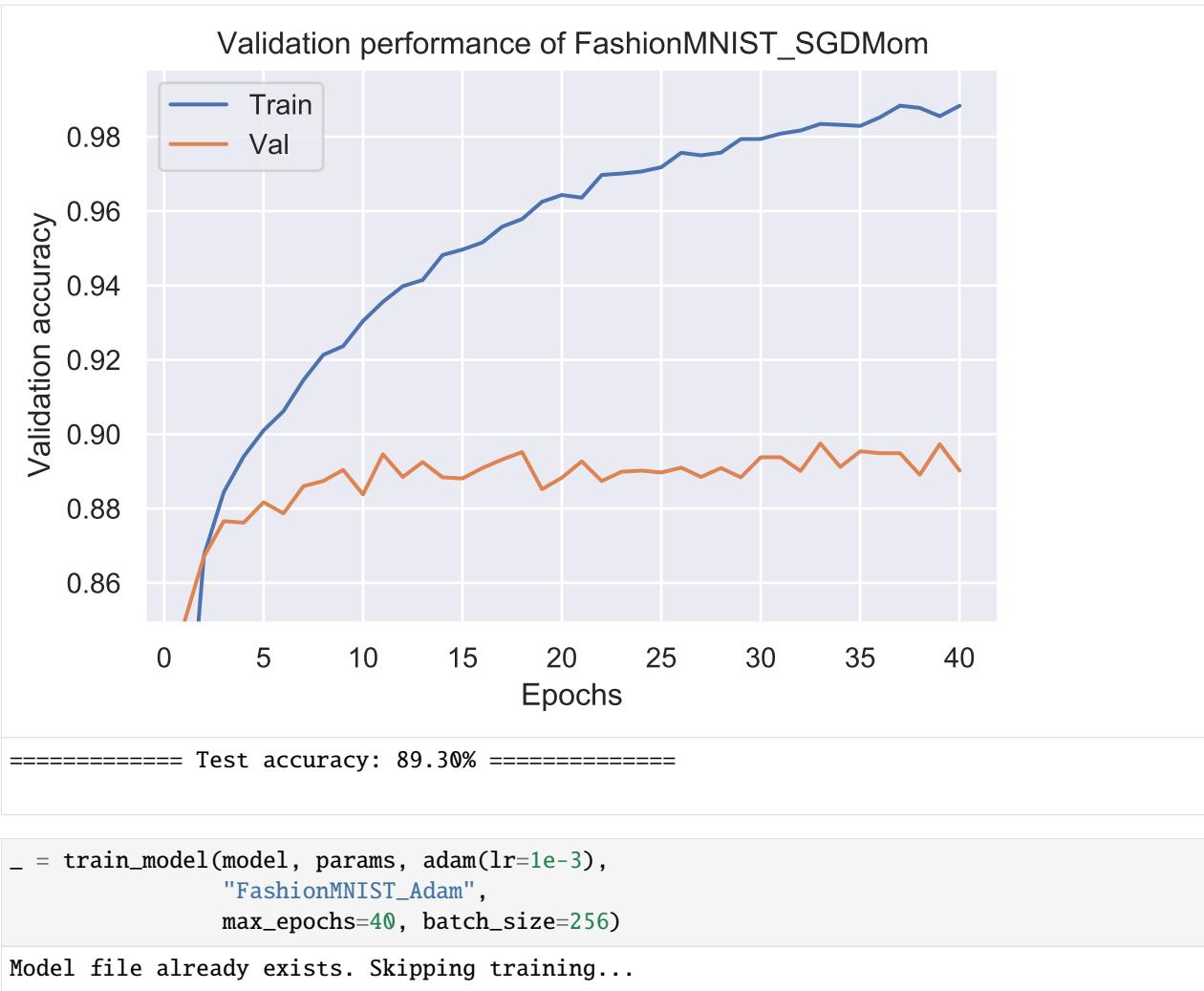
[23]: model, params = init_simple_model(kaiming_init, act_fn=nn.relu)

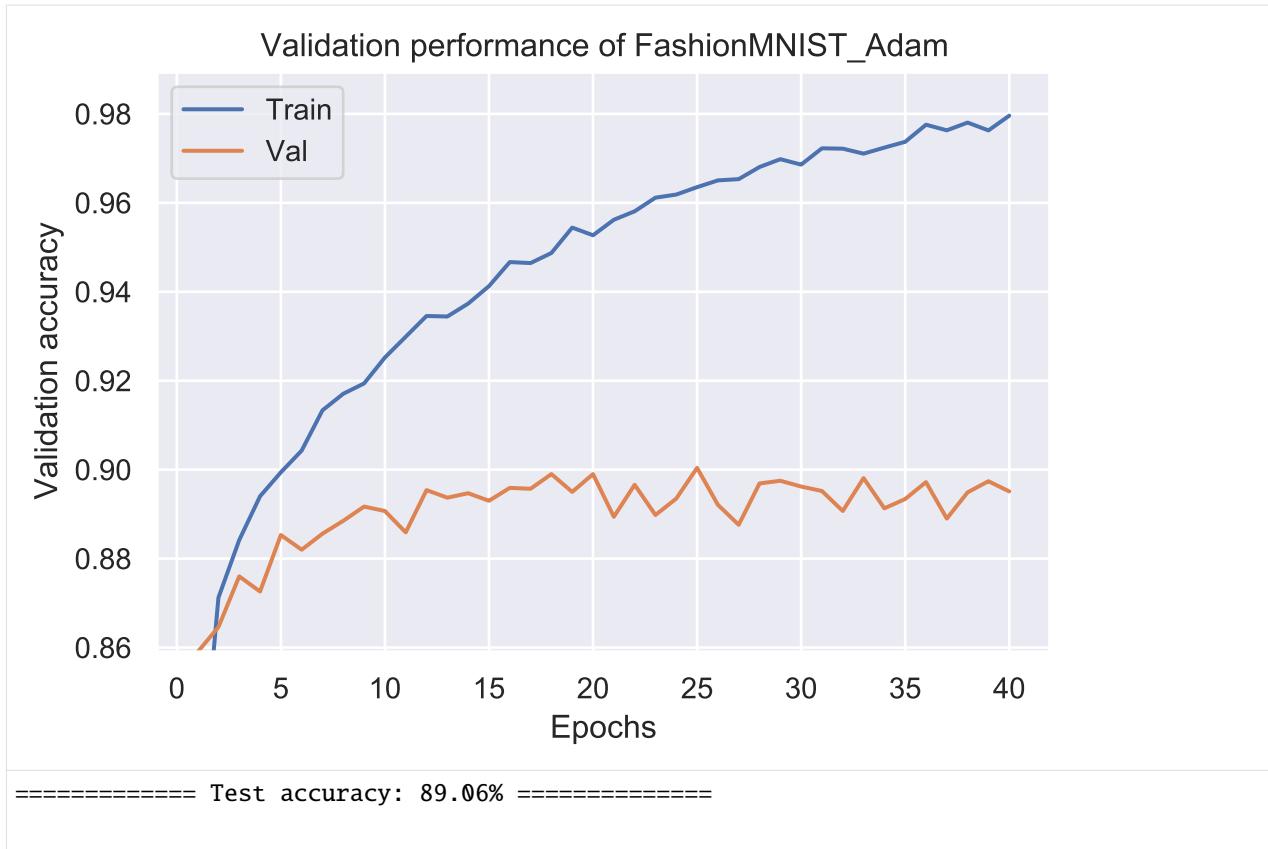
For a fair comparison, we train the exact same model with the same seed with the three optimizers below. Feel free to change the hyperparameters if you want, the models are relatively fast to train.

[24]: _ = train_model(model, params, sgd(lr=1e-1),
 "FashionMNIST_SGD",
 max_epochs=40, batch_size=256)

Model file already exists. Skipping training...







The result is that all optimizers perform similarly well with the given model. The differences are too small to find any significant conclusion. However, keep in mind that this can also be attributed to the initialization we chose. When changing the initialization to worse (e.g. constant variance), Adam usually shows to be more robust because of its adaptive learning rate. To show the specific benefits of the optimizers, we will continue to look at some possible loss surfaces in which momentum and adaptive learning rate are crucial.

Pathological curvatures

A pathological curvature is a type of surface that is similar to ravines and is particularly tricky for plain SGD optimization. In words, pathological curvatures typically have a steep gradient in one direction with an optimum at the center, while in a second direction we have a slower gradient towards a (global) optimum. Let's first create an example surface of this and visualize it:

```
[27]: def pathological_curve_loss(w1, w2):
    # Example of a pathological curvature. There are many more possible, feel free to
    ↪experiment here!
    x1_loss = nn.tanh(w1)**2 + 0.01 * jnp.abs(w1)
    x2_loss = nn.sigmoid(w2)
    return x1_loss + x2_loss
```

```
[28]: def plot_curve(curve_fn, x_range=(-5,5), y_range=(-5,5), plot_3d=False, cmap=cm.viridis, ↪
    title="Pathological curvature"):
    fig = plt.figure()
    ax = plt.axes(projection='3d') if plot_3d else plt.axes()
```

(continues on next page)

(continued from previous page)

```

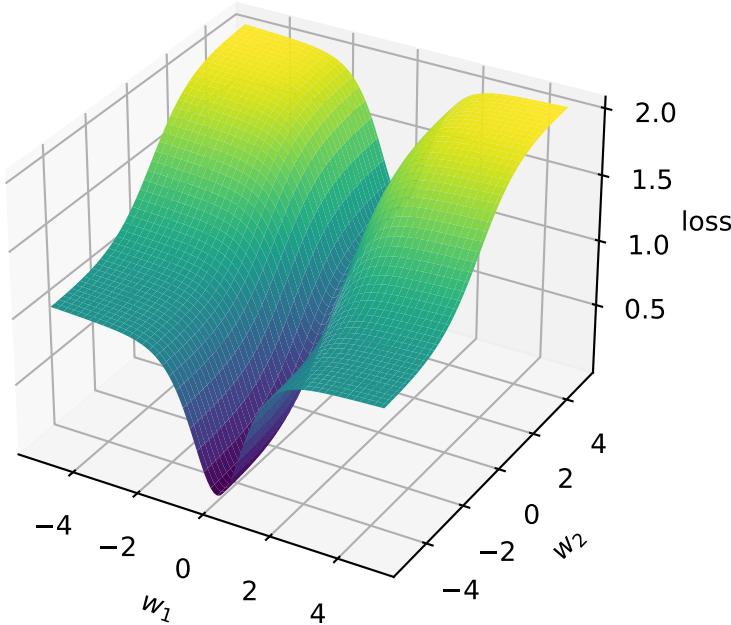
x = np.arange(x_range[0], x_range[1], (x_range[1]-x_range[0])/100.)
y = np.arange(y_range[0], y_range[1], (y_range[1]-y_range[0])/100.)
x, y = np.meshgrid(x, y)
z = curve_fn(x, y)
z = jax.device_get(z)

if plot_3d:
    ax.plot_surface(x, y, z, cmap=cmap, linewidth=1, color="#000", antialiased=False)
    ax.set_zlabel("loss")
else:
    ax.imshow(z[::-1], cmap=cmap, extent=(x_range[0], x_range[1], y_range[0], y_range[1]))
    plt.title(title)
    ax.set_xlabel(r"$w_1$")
    ax.set_ylabel(r"$w_2$")
    plt.tight_layout()
return ax

sns.reset_orig()
_ = plot_curve(pathological_curve_loss, plot_3d=True)
plt.show()

```

Pathological curvature



In terms of optimization, you can image that w_1 and w_2 are weight parameters, and the curvature represents the loss surface over the space of w_1 and w_2 . Note that in typical networks, we have many, many more parameters than two, and such curvatures can occur in multi-dimensional spaces as well.

Ideally, our optimization algorithm would find the center of the ravine and focuses on optimizing the parameters towards the direction of w_2 . However, if we encounter a point along the ridges, the gradient is much greater in w_1 than w_2 , and we might end up jumping from one side to the other. Due to the large gradients, we would have to reduce our learning

rate slowing down learning significantly.

To test our algorithms, we can implement a simple function to train two parameters on such a surface:

```
[29]: def train_curve(optimizer, curve_func=pathological_curve_loss, num_updates=100, init=[5,
→ 5]):
    """
    Inputs:
        optimizer - Optimizer to use
        curve_func - Loss function (e.g. pathological curvature)
        num_updates - Number of updates/steps to take when optimizing
        init - Initial values of parameters. Must be a list/tuple with two elements,
→ representing w_1 and w_2
    Outputs:
        Numpy array of shape [num_updates, 3] with [t,:2] being the parameter values at
→ step t, and [t,2] the loss at t.
    """
    weights = jnp.array(init, dtype=jnp.float32)
    grad_fn = jax.jit(jax.value_and_grad(lambda w: curve_func(w[0], w[1])))
    opt_state = optimizer.init(weights)

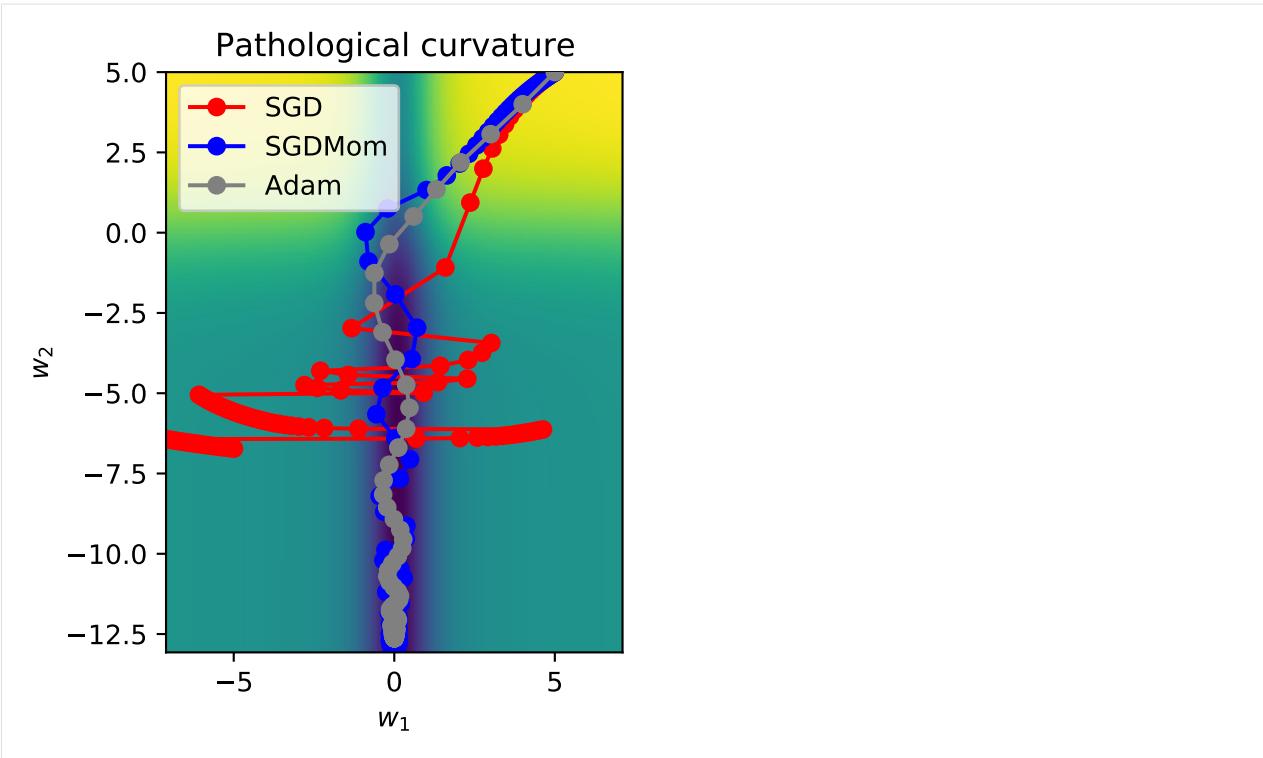
    list_points = []
    for _ in range(num_updates):
        loss, grads = grad_fn(weights)
        list_points.append(jnp.concatenate([weights, loss[None]], axis=0))
        updates, opt_state = optimizer.update(grads, opt_state)
        weights = optax.apply_updates(weights, updates)
    points = jnp.stack(list_points, axis=0)
    points = jax.device_get(points)
    return points
```

Next, let's apply the different optimizers on our curvature. Note that we set a much higher learning rate for the optimization algorithms as you would in a standard neural network. This is because we only have 2 parameters instead of tens of thousands or even millions.

```
[30]: SGD_points = train_curve(sgd(lr=10))
SGDMom_points = train_curve(sgd_momentum(lr=10, momentum=0.9))
Adam_points = train_curve(adam(lr=1))
```

To understand best how the different algorithms worked, we visualize the update step as a line plot through the loss surface. We will stick with a 2D representation for readability.

```
[31]: all_points = np.concatenate([SGD_points, SGDMom_points, Adam_points], axis=0)
ax = plot_curve(pathological_curve_loss,
                 x_range=(-np.absolute(all_points[:,0]).max(), np.absolute(all_points[:,
→ 0]).max()),
                 y_range=(all_points[:,1].min(), all_points[:,1].max()),
                 plot_3d=False)
ax.plot(SGD_points[:,0], SGD_points[:,1], color="red", marker="o", zorder=1, label="SGD")
ax.plot(SGDMom_points[:,0], SGDMom_points[:,1], color="blue", marker="o", zorder=2,
→ label="SGDMom")
ax.plot(Adam_points[:,0], Adam_points[:,1], color="grey", marker="o", zorder=3, label=
→ "Adam")
plt.legend()
plt.show()
```



We can clearly see that SGD is not able to find the center of the optimization curve and has a problem converging due to the steep gradients in w_1 . In contrast, Adam and SGD with momentum nicely converge as the changing direction of w_1 is canceling itself out. On such surfaces, it is crucial to use momentum.

Steep optima

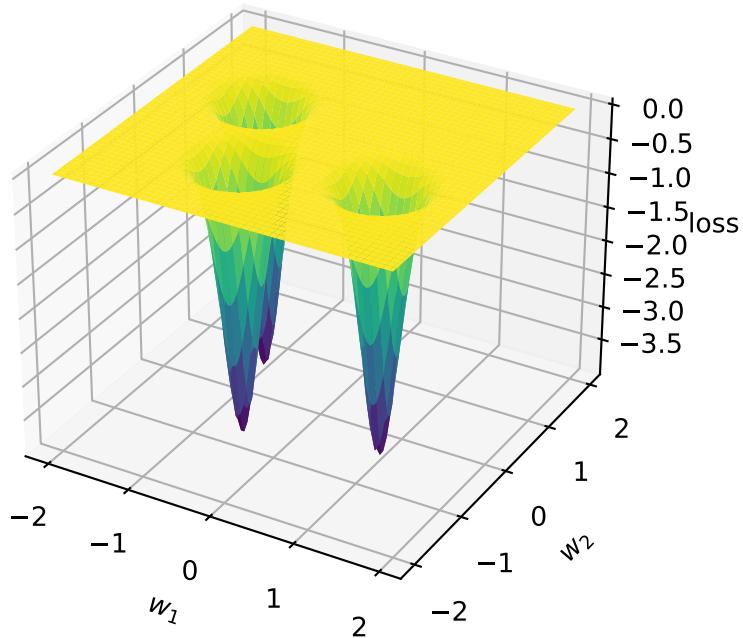
A second type of challenging loss surfaces are steep optima. In those, we have a larger part of the surface having very small gradients while around the optimum, we have very large gradients. For instance, take the following loss surfaces:

```
[32]: def bivar_gaussian(w1, w2, x_mean=0.0, y_mean=0.0, x_sig=1.0, y_sig=1.0):
    norm = 1 / (2 * np.pi * x_sig * y_sig)
    x_exp = (-1 * (w1 - x_mean)**2) / (2 * x_sig**2)
    y_exp = (-1 * (w2 - y_mean)**2) / (2 * y_sig**2)
    return norm * jnp.exp(x_exp + y_exp)

def comb_func(w1, w2):
    z = -bivar_gaussian(w1, w2, x_mean=1.0, y_mean=-0.5, x_sig=0.2, y_sig=0.2)
    z -= bivar_gaussian(w1, w2, x_mean=-1.0, y_mean=0.5, x_sig=0.2, y_sig=0.2)
    z -= bivar_gaussian(w1, w2, x_mean=-0.5, y_mean=-0.8, x_sig=0.2, y_sig=0.2)
    return z

_ = plot_curve(comb_func, x_range=(-2, 2), y_range=(-2, 2), plot_3d=True, title="Steep ↴ optima")
```

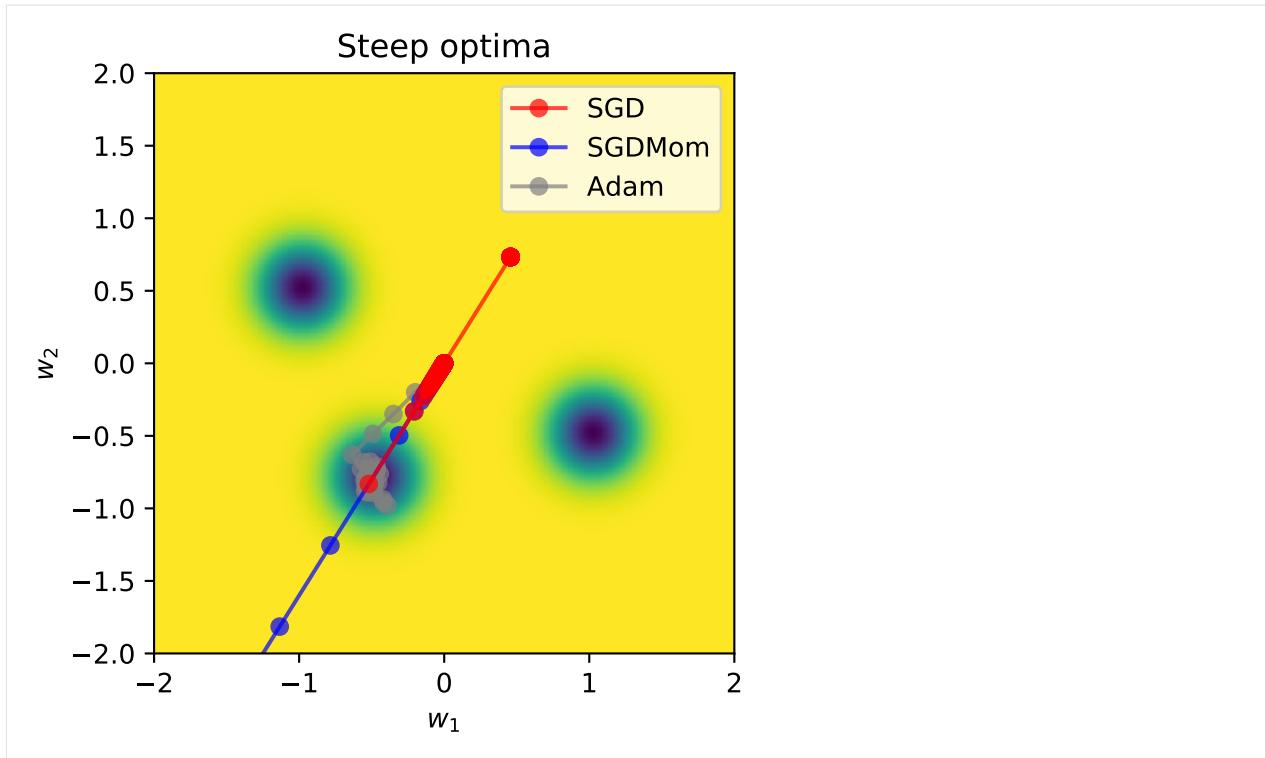
Steep optima



Most of the loss surface has very little to no gradients. However, close to the optima, we have very steep gradients. To reach the minimum when starting in a region with lower gradients, we expect an adaptive learning rate to be crucial. To verify this hypothesis, we can run our three optimizers on the surface:

```
[33]: SGD_points = train_curve(sgd(lr=.5), comb_func, init=[0,0])
SGDMom_points = train_curve(sgd_momentum(lr=1, momentum=0.9), comb_func, init=[0,0])
Adam_points = train_curve(adam(lr=0.2), comb_func, init=[0,0])

all_points = np.concatenate([SGD_points, SGDMom_points, Adam_points], axis=0)
ax = plot_curve(comb_func,
                 x_range=(-2, 2),
                 y_range=(-2, 2),
                 plot_3d=False,
                 title="Steep optima")
ax.plot(SGD_points[:,0], SGD_points[:,1], color="red", marker="o", zorder=3, label="SGD",
        alpha=0.7)
ax.plot(SGDMom_points[:,0], SGDMom_points[:,1], color="blue", marker="o", zorder=2,
        label="SGDMom", alpha=0.7)
ax.plot(Adam_points[:,0], Adam_points[:,1], color="grey", marker="o", zorder=1, label=
        "Adam", alpha=0.7)
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
plt.legend()
plt.show()
```



SGD first takes very small steps until it touches the border of the optimum. First reaching a point around $(-0.75, -0.5)$, the gradient direction has changed and pushes the parameters to $(0.8, 0.5)$ from which SGD cannot recover anymore (only with many, many steps). A similar problem has SGD with momentum, only that it continues the direction of the touch of the optimum. The gradients from this time step are so much larger than any other point that the momentum m_t is overpowered by it. Finally, Adam is able to converge in the optimum showing the importance of adaptive learning rates.

What optimizer to take

After seeing the results on optimization, what is our conclusion? Should we always use Adam and never look at SGD anymore? The short answer: no. There are many papers saying that in certain situations, SGD (with momentum) generalizes better where Adam often tends to overfit [5,6]. This is related to the idea of finding wider optima. For instance, see the illustration of different optima below (credit: [Keskar et al., 2017](#)):

The black line represents the training loss surface, while the dotted red line is the test loss. Finding sharp, narrow minima can be helpful for finding the minimal training loss. However, this doesn't mean that it also minimizes the test loss as especially flat minima have shown to generalize better. You can imagine that the test dataset has a slightly shifted loss surface due to the different examples than in the training set. A small change can have a significant influence for sharp minima, while flat minima are generally more robust to this change.

In the next tutorial, we will see that some network types can still be better optimized with SGD and learning rate scheduling than Adam. Nevertheless, Adam is the most commonly used optimizer in Deep Learning as it usually performs better than other optimizers, especially for deep networks.

4.21.4 Conclusion

In this tutorial, we have looked at initialization and optimization techniques for neural networks. We have seen that a good initialization has to balance the preservation of the gradient variance as well as the activation variance. This can be achieved with the Xavier initialization for tanh-based networks, and the Kaiming initialization for ReLU-based networks. In optimization, concepts like momentum and adaptive learning rate can help with challenging loss surfaces but don't guarantee an increase in performance for neural networks.

4.21.5 References

- [1] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010. [link](#)
 - [2] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE international conference on computer vision. 2015. [link](#)
 - [3] Kingma, Diederik P. & Ba, Jimmy. "Adam: A Method for Stochastic Optimization." Proceedings of the third international conference for learning representations (ICLR). 2015. [link](#)
 - [4] Keskar, Nitish Shirish, et al. "On large-batch training for deep learning: Generalization gap and sharp minima." Proceedings of the fifth international conference for learning representations (ICLR). 2017. [link](#)
 - [5] Wilson, Ashia C., et al. "The Marginal Value of Adaptive Gradient Methods in Machine Learning." Advances in neural information processing systems. 2017. [link](#)
 - [6] Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint. 2017. [link](#)
-

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.22 Tutorial 5 (JAX): Inception, ResNet and DenseNet

Filled notebook:

Pre-trained models:

PyTorch version:

Author: Phillip Lippe

Note: This notebook is written in JAX+Flax. It is a 1-to-1 translation of the original notebook written in PyTorch+PyTorch Lightning with almost identical results. For an introduction to JAX, check out our [Tutorial 2 \(JAX\): Introduction to JAX+Flax](#). Further, throughout the notebook, we comment on major differences to the PyTorch version and provide explanations for the major parts of the JAX code.

Speed comparison: We note the training times for all models in the PyTorch and the JAX implementation below (PyTorch v1.11, JAX v0.3.13). The models were trained on the same hardware (NVIDIA RTX3090, 24 core CPU) and we slightly adjusted the tutorials to use the exact same training settings (200 epochs, data loading parameters, evaluation schedule, etc.). Overall, the JAX implementation is about *2.5-3.4x faster* than PyTorch! However, with larger models, larger batch sizes, or smaller GPUs, the speed up is expected to become considerably smaller.

Models	PyTorch	JAX
GoogleNet	53min 50sec	16min 10sec
ResNet	20min 47sec	7min 51sec
Pre-Activation ResNet	20min 57sec	8min 25sec
DenseNet	49min 23sec	20min 1sec

In this tutorial, we will implement and discuss variants of modern CNN architectures. There have been many different architectures been proposed over the past few years. Some of the most impactful ones, and still relevant today, are the following: [GoogleNet](#)/Inception architecture (winner of ILSVRC 2014), [ResNet](#) (winner of ILSVRC 2015), and [DenseNet](#) (best paper award CVPR 2017). All of them were state-of-the-art models when being proposed, and the core ideas of these networks are the foundations for most current state-of-the-art architectures. Thus, it is important to understand these architectures in detail and learn how to implement them.

Let's start with importing our standard libraries here.

```
[1]: ## Standard libraries
import os
import numpy as np
from PIL import Image
from typing import Any
from collections import defaultdict
import time

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## Progress bar
from tqdm.auto import tqdm

## To run JAX on TPU in Google Colab, uncomment the two lines below
# import jax.tools.colab_tpu
# jax.tools.colab_tpu.setup_tpu()

## JAX
import jax
import jax.numpy as jnp
from jax import random

## Flax (NN in JAX)
```

(continues on next page)

(continued from previous page)

```

try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax
from flax import linen as nn
from flax.training import train_state, checkpoints

## Optax (Optimizers in JAX)
try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax

## PyTorch
import torch
import torch.utils.data as data
from torch.utils.tensorboard import SummaryWriter
import torchvision
from torchvision import transforms
from torchvision.datasets import CIFAR10

```

We will use the same path variables DATASET_PATH and CHECKPOINT_PATH as in the previous tutorials. Adjust the paths if necessary.

```
[2]: # Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../../saved_models/tutorial5_jax"

# Seeding for random operations
main_rng = random.PRNGKey(42)

print("Device:", jax.devices()[0])
Device: gpu:0
```

We also have pretrained models and TensorBoards (more on this later) for this tutorial, and download them below.

```
[3]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/JAX/tutorial5/"
# Files to download
pretrained_files = ["GoogleNet.ckpt", "ResNet.ckpt", "PreActResNet.ckpt", "DenseNet.ckpt",
                    "tensorboards/GoogleNet/events.out.tfevents.googlenet",
                    "tensorboards/ResNet/events.out.tfevents.resnet",
                    "tensorboards/PreActResNet/events.out.tfevents.preactresnet",
                    "tensorboards/DenseNet/events.out.tfevents.densenet"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)
```

(continues on next page)

(continued from previous page)

```
# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive_"
                  "folder, or contact the author with the full output including the following error:\n",
                  e)
```

Throughout this tutorial, we will train and evaluate the models on the CIFAR10 dataset. This allows you to compare the results obtained here with the model you have implemented in the first assignment. As we have learned from the previous tutorial about initialization, it is important to have the data preprocessed with a zero mean. Therefore, as a first step, we will calculate the mean and standard deviation of the CIFAR dataset:

```
[4]: train_dataset = CIFAR10(root=DATASET_PATH, train=True, download=True)
DATA_MEANS = (train_dataset.data / 255.0).mean(axis=(0,1,2))
DATA_STD = (train_dataset.data / 255.0).std(axis=(0,1,2))
print("Data mean", DATA_MEANS)
print("Data std", DATA_STD)

Files already downloaded and verified
Data mean [0.49139968 0.48215841 0.44653091]
Data std [0.24703223 0.24348513 0.26158784]
```

We will use this information to normalize our data accordingly. Additionally, we will use transformations from the package `torchvision` to implement data augmentations during training. This reduces the risk of overfitting and helps CNNs to generalize better. Specifically, we will apply two random augmentations.

First, we will flip each image horizontally by a chance of 50% (`transforms.RandomHorizontalFlip`). The object class usually does not change when flipping an image, and we don't expect any image information to be dependent on the horizontal orientation. This would be however different if we would try to detect digits or letters in an image, as those have a certain orientation.

The second augmentation we use is called `transforms.RandomResizedCrop`. This transformation scales the image in a small range, while eventually changing the aspect ratio, and crops it afterward in the previous size. Therefore, the actual pixel values change while the content or overall semantics of the image stays the same.

We will randomly split the training dataset into a training and a validation set. The validation set will be used for determining early stopping. After finishing the training, we test the models on the CIFAR test set.

```
[5]: # Transformations applied on each image => bring them into a numpy array
def image_to_numpy(img):
    img = np.array(img, dtype=np.float32)
    img = (img / 255. - DATA_MEANS) / DATA_STD
    return img

# We need to stack the batch elements
```

(continues on next page)

(continued from previous page)

```

def numpy_collate(batch):
    if isinstance(batch[0], np.ndarray):
        return np.stack(batch)
    elif isinstance(batch[0], (tuple, list)):
        transposed = zip(*batch)
        return [numpy_collate(samples) for samples in transposed]
    else:
        return np.array(batch)

test_transform = image_to_numpy
# For training, we add some augmentation. Networks are too powerful and would overfit.
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                      transforms.RandomResizedCrop((32, 32), scale=(0.8, 1.
                                          ↪0)), ratio=(0.9, 1.1)),
                                      image_to_numpy
                                      ])
# Loading the training dataset. We need to split it into a training and validation part
# We need to do a little trick because the validation set should not use the
# ↪augmentation.
train_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=train_transform,
                        ↪download=True)
val_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=test_transform,
                      ↪download=True)
train_set, _ = torch.utils.data.random_split(train_dataset, [45000, 5000],
                                             ↪generator=torch.Generator().manual_seed(42))
_, val_set = torch.utils.data.random_split(val_dataset, [45000, 5000], generator=torch.
                                             ↪Generator().manual_seed(42))

# Loading the test set
test_set = CIFAR10(root=DATASET_PATH, train=False, transform=test_transform,
                    ↪download=True)

# We define a set of data loaders that we can use for training and validation
train_loader = data.DataLoader(train_set,
                                batch_size=128,
                                shuffle=True,
                                drop_last=True,
                                collate_fn=numpy_collate,
                                num_workers=8,
                                persistent_workers=True)
val_loader = data.DataLoader(val_set,
                             batch_size=128,
                             shuffle=False,
                             drop_last=False,
                             collate_fn=numpy_collate,
                             num_workers=4,
                             persistent_workers=True)
test_loader = data.DataLoader(test_set,
                             batch_size=128,
                             shuffle=False,
                             drop_last=False,

```

(continues on next page)

(continued from previous page)

```
collate_fn=numpy_collate,
num_workers=4,
persistent_workers=True)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

To verify that our normalization works, we can print out the mean and standard deviation of the single batch. The mean should be close to 0 and the standard deviation close to 1 for each channel:

```
[6]: imgs, _ = next(iter(train_loader))
print("Batch mean", imgs.mean(axis=(0,1,2)))
print("Batch std", imgs.std(axis=(0,1,2)))

Batch mean [-0.05225317 -0.06281322 -0.11905593]
Batch std [0.95375352 0.92699525 0.91463391]
```

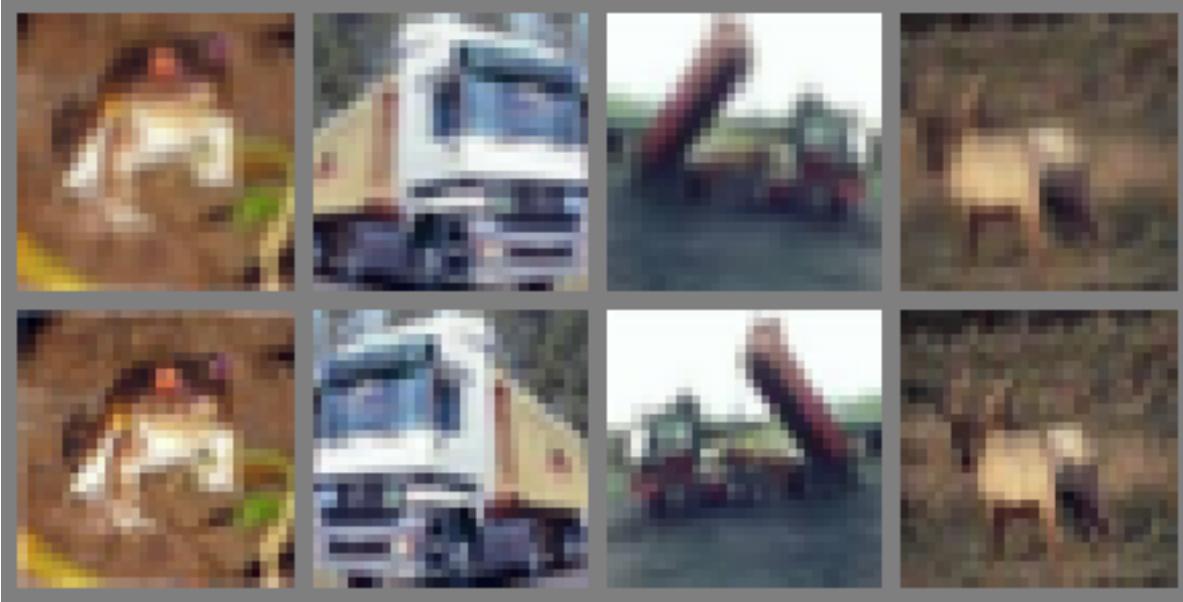
Finally, let's visualize a few images from the training set, and how they look like after random data augmentation:

```
[7]: NUM_IMAGES = 4
images = [train_dataset[idx] for idx in range(NUM_IMAGES)]
orig_images = [Image.fromarray(train_dataset.data[idx]) for idx in range(NUM_IMAGES)]
orig_images = [test_transform(img) for img in orig_images]

imgs = np.stack(images + orig_images, axis=0)
imgs = torch.from_numpy(imgs).permute(0, 3, 1, 2)
img_grid = torchvision.utils.make_grid(imgs, nrow=4, normalize=True, pad_value=0.5)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("Augmentation examples on CIFAR10 (Top: augmented, bottom: original)")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

Augmentation examples on CIFAR10 (Top: augmented, bottom: original)



4.22.1 Trainer Module

In the [PyTorch version](#) of this tutorial, we would now introduce the framework [PyTorch Lightning](#) which simplifies the overall training of a model. So far (June 2022), there is no clear equivalent of it for JAX. Some basic training functionalities are implemented in [flax.training](#) ([documentation](#)), and predefined training modules are implemented in [trax](#) ([documentation](#)), but neither provide a complete, flexible training package yet like PyTorch Lightning. Hence, we need to write our own small training loop.

For this, we take inspiration from PyTorch Lightning and build a trainer module/object with the following main functionalities:

1. *Storing model and parameters*: In order to train multiple models with different hyperparameters, the trainer module creates an instance of the model class, and keeps the parameters in the same class. This way, we can easily apply a model with its parameters on new inputs.
2. *Initialization of model and training state*: During initialization of the trainer, we initialize the model parameters and a new train state, which includes the optimizer and possible learning rate schedulers.
3. *Training, validation and test loops*: Similar to PyTorch Lightning, we implement simple training, validation and test loops, where subclasses of this trainer could overwrite the respective training, validation or test steps. Since in this tutorial, all models will have the same objective, i.e. classification on CIFAR10, we will pre-specify them in the trainer module below.
4. *Logging, saving and loading of models*: To keep track of the training, we implement functionalities to log the training progress and save the best model on the validation set. Afterwards, this model can be loaded from the disk.

Before starting to implement a trainer module with these functionalities, we need to take one prior step. The networks we will implement in this tutorial use BatchNormalization, which carries an exponential average of the prior batch statistics (mean and std) to apply during evaluation. In PyTorch, this is simply tracked by an object attribute of an object of the class `nn.BatchNorm2d`, but in JAX, we only work with functions. Hence, we need to take care of the batch statistics ourselves, similar to the parameters, and enter them during every forward pass. To simplify this a little, we overwrite the `train_state.TrainState` class of Flax by adding a field for the batch statistics:

```
[8]: class TrainState(train_state.TrainState):
    # A simple extension of TrainState to also include batch statistics
    batch_stats: Any
```

With this, the training state contains both the training parameters and the batch statistics, which makes it easier to keep everything in one place.

Now that the batch statistics are sorted out, we can implement our full training module:

```
[9]: class TrainerModule:

    def __init__(self,
                 model_name : str,
                 model_class : nn.Module,
                 model_hparams : dict,
                 optimizer_name : str,
                 optimizer_hparams : dict,
                 exmp_imgs : Any,
                 seed=42):
        """
        Module for summarizing all training functionalities for classification on
        ↵CIFAR10.

        Inputs:
            model_name - String of the class name, used for logging and saving
            model_class - Class implementing the neural network
            model_hparams - Hyperparameters of the model, used as input to model
        ↵constructor
            optimizer_name - String of the optimizer name, supporting ['sgd', 'adam', 'adamw'
        ↵']
            optimizer_hparams - Hyperparameters of the optimizer, including learning
        ↵rate as 'lr'
            exmp_imgs - Example imgs, used as input to initialize the model
            seed - Seed to use in the model initialization
        """
        super().__init__()
        self.model_name = model_name
        self.model_class = model_class
        self.model_hparams = model_hparams
        self.optimizer_name = optimizer_name
        self.optimizer_hparams = optimizer_hparams
        self.seed = seed
        # Create empty model. Note: no parameters yet
        self.model = self.model_class(**self.model_hparams)
        # Prepare logging
        self.log_dir = os.path.join(CHECKPOINT_PATH, self.model_name)
        self.logger = SummaryWriter(log_dir=self.log_dir)
        # Create jitted training and eval functions
        self.create_functions()
        # Initialize model
        self.init_model(exmp_imgs)

    def create_functions(self):
```

(continues on next page)

(continued from previous page)

```

# Function to calculate the classification loss and accuracy for a model
def calculate_loss(params, batch_stats, batch, train):
    imgs, labels = batch
    # Run model. During training, we need to update the BatchNorm statistics.
    outs = self.model.apply({'params': params, 'batch_stats': batch_stats},
                           imgs,
                           train=train,
                           mutable=['batch_stats'] if train else False)
    logits, new_model_state = outs if train else (outs, None)
    loss = optax.softmax_cross_entropy_with_integer_labels(logits, labels).mean()
    acc = (logits.argmax(axis=-1) == labels).mean()
    return loss, (acc, new_model_state)

# Training function
def train_step(state, batch):
    loss_fn = lambda params: calculate_loss(params, state.batch_stats, batch,
                                           train=True)
    # Get loss, gradients for loss, and other outputs of loss function
    ret, grads = jax.value_and_grad(loss_fn, has_aux=True)(state.params)
    loss, acc, new_model_state = ret[0], *ret[1]
    # Update parameters and batch statistics
    state = state.apply_gradients(grads=grads, batch_stats=new_model_state[
        'batch_stats'])
    return state, loss, acc

# Eval function
def eval_step(state, batch):
    # Return the accuracy for a single batch
    _, (acc, _) = calculate_loss(state.params, state.batch_stats, batch,
                                 train=False)
    return acc

# jit for efficiency
self.train_step = jax.jit(train_step)
self.eval_step = jax.jit(eval_step)

def init_model(self, exmp_imgs):
    # Initialize model
    init_rng = jax.random.PRNGKey(self.seed)
    variables = self.model.init(init_rng, exmp_imgs, train=True)
    self.init_params, self.init_batch_stats = variables['params'], variables['batch_
    stats']
    self.state = None

def init_optimizer(self, num_epochs, num_steps_per_epoch):
    # Initialize learning rate schedule and optimizer
    if self.optimizer_name.lower() == 'adam':
        opt_class = optax.adam
    elif self.optimizer_name.lower() == 'adamw':
        opt_class = optax.adamw
    elif self.optimizer_name.lower() == 'sgd':
        opt_class = optax.sgd
    else:
        assert False, f'Unknown optimizer "{opt_class}"'
    # We decrease the learning rate by a factor of 0.1 after 60% and 85% of the_
    training

```

(continues on next page)

(continued from previous page)

```

lr_schedule = optax.piecewise_constant_schedule(
    init_value=self.optimizer_hparams.pop('lr'),
    boundaries_and_scales=
        {int(num_steps_per_epoch*num_epochs*0.6): 0.1,
         int(num_steps_per_epoch*num_epochs*0.85): 0.1}
)
# Clip gradients at max value, and evt. apply weight decay
transf = [optax.clip(1.0)]
if opt_class == optax.sgd and 'weight_decay' in self.optimizer_hparams: # wd is
→ integrated in adamw
    transf.append(optax.add_decayed_weights(self.optimizer_hparams.pop('weight_
→ decay'))))
optimizer = optax.chain(
    *transf,
    opt_class(lr_schedule, **self.optimizer_hparams)
)
# Initialize training state
self.state = TrainState.create(apply_fn=self.model.apply,
                               params=self.init_params if self.state is None_
→ else self.state.params,
                               batch_stats=self.init_batch_stats if self.state_
→ is None else self.state.batch_stats,
                               tx=optimizer)

def train_model(self, train_loader, val_loader, num_epochs=200):
    # Train model for defined number of epochs
    # We first need to create optimizer and the scheduler for the given number of_
→ epochs
    self.init_optimizer(num_epochs, len(train_loader))
    # Track best eval accuracy
    best_eval = 0.0
    for epoch_idx in tqdm(range(1, num_epochs+1)):
        self.train_epoch(train_loader, epoch=epoch_idx)
        if epoch_idx % 2 == 0:
            eval_acc = self.eval_model(val_loader)
            self.logger.add_scalar('val/acc', eval_acc, global_step=epoch_idx)
            if eval_acc >= best_eval:
                best_eval = eval_acc
                self.save_model(step=epoch_idx)
                self.logger.flush()

def train_epoch(self, train_loader, epoch):
    # Train model for one epoch, and log avg loss and accuracy
    metrics = defaultdict(list)
    for batch in tqdm(train_loader, desc='Training', leave=False):
        self.state, loss, acc = self.train_step(self.state, batch)
        metrics['loss'].append(loss)
        metrics['acc'].append(acc)
    for key in metrics:
        avg_val = np.stack(jax.device_get(metrics[key])).mean()
        self.logger.add_scalar('train/'+key, avg_val, global_step=epoch)

```

(continues on next page)

(continued from previous page)

```

def eval_model(self, data_loader):
    # Test model on all images of a data loader and return avg loss
    correct_class, count = 0, 0
    for batch in data_loader:
        acc = self.eval_step(self.state, batch)
        correct_class += acc * batch[0].shape[0]
        count += batch[0].shape[0]
    eval_acc = (correct_class / count).item()
    return eval_acc

def save_model(self, step=0):
    # Save current model at certain training iteration
    checkpoints.save_checkpoint(ckpt_dir=self.log_dir,
                                target={'params': self.state.params,
                                         'batch_stats': self.state.batch_stats},
                                step=step,
                                overwrite=True)

def load_model(self, pretrained=False):
    # Load model. We use different checkpoint for pretrained models
    if not pretrained:
        state_dict = checkpoints.restore_checkpoint(ckpt_dir=self.log_dir,
                                                target=None)
    else:
        state_dict = checkpoints.restore_checkpoint(ckpt_dir=os.path.join(CHECKPOINT_PATH,
                                                                f'{self.model_name}.ckpt'), target=None)
        self.state = TrainState.create(apply_fn=self.model.apply,
                                      params=state_dict['params'],
                                      batch_stats=state_dict['batch_stats'],
                                      tx=self.state.tx if self.state else optax.sgd(0.
                                         1) # Default optimizer
                                         )
    return state_dict

def checkpoint_exists(self):
    # Check whether a pretrained model exist for this autoencoder
    return os.path.isfile(os.path.join(CHECKPOINT_PATH, f'{self.model_name}.ckpt'))

```

Next, we can use this trainer module to create a compact training function:

```
[10]: def train_classifier(*args, num_epochs=200, **kwargs):
    # Create a trainer module with specified hyperparameters
    trainer = TrainerModule(*args, **kwargs)
    if not trainer.checkpoint_exists(): # Skip training if pretrained model exists
        trainer.train_model(train_loader, val_loader, num_epochs=num_epochs)
        trainer.load_model()
    else:
        trainer.load_model(pretrained=True)
    # Test trained model
    val_acc = trainer.eval_model(val_loader)
    test_acc = trainer.eval_model(test_loader)
    return trainer, {'val': val_acc, 'test': test_acc}
```

Finally, we can focus on the Convolutional Neural Networks we want to implement today: GoogleNet, ResNet, and

DenseNet.

4.22.2 Inception

The [GoogleNet](#), proposed in 2014, won the ImageNet Challenge because of its usage of the Inception modules. In general, we will mainly focus on the concept of Inception in this tutorial instead of the specifics of the GoogleNet, as based on Inception, there have been many follow-up works ([Inception-v2](#), [Inception-v3](#), [Inception-v4](#), [Inception-ResNet](#),...). The follow-up works mainly focus on increasing efficiency and enabling very deep Inception networks. However, for a fundamental understanding, it is sufficient to look at the original Inception block.

An Inception block applies four convolution blocks separately on the same feature map: a 1x1, 3x3, and 5x5 convolution, and a max pool operation. This allows the network to look at the same data with different receptive fields. Of course, learning only 5x5 convolution would be theoretically more powerful. However, this is not only more computation and memory heavy but also tends to overfit much easier. The overall inception block looks like below (figure credit - [Szegedy et al.](#)):

The additional 1x1 convolutions before the 3x3 and 5x5 convolutions are used for dimensionality reduction. This is especially crucial as the feature maps of all branches are merged afterward, and we don't want any explosion of feature size. As 5x5 convolutions are 25 times more expensive than 1x1 convolutions, we can save a lot of computation and parameters by reducing the dimensionality before the large convolutions.

We can now try to implement the Inception Block ourselves:

```
[11]: googlenet_kernel_init = nn.initializers.kaiming_normal()

class InceptionBlock(nn.Module):
    c_red : dict # Dictionary of reduced dimensionalities with keys "1x1", "3x3", "5x5",
    ↵ and "max"
    c_out : dict # Dictionary of output feature sizes with keys "1x1", "3x3", "5x5",
    ↵ and "max"
    act_fn : callable # Activation function

    @nn.compact
    def __call__(self, x, train=True):
        # 1x1 convolution branch
        x_1x1 = nn.Conv(self.c_out["1x1"], kernel_size=(1, 1), kernel_init=googlenet_
        ↵ kernel_init, use_bias=False)(x)
        x_1x1 = nn.BatchNorm()(x_1x1, use_running_average=not train)
        x_1x1 = self.act_fn(x_1x1)

        # 3x3 convolution branch
        x_3x3 = nn.Conv(self.c_red["3x3"], kernel_size=(1, 1), kernel_init=googlenet_
        ↵ kernel_init, use_bias=False)(x)
        x_3x3 = nn.BatchNorm()(x_3x3, use_running_average=not train)
        x_3x3 = self.act_fn(x_3x3)
        x_3x3 = nn.Conv(self.c_out["3x3"], kernel_size=(3, 3), kernel_init=googlenet_
        ↵ kernel_init, use_bias=False)(x_3x3)
        x_3x3 = nn.BatchNorm()(x_3x3, use_running_average=not train)
        x_3x3 = self.act_fn(x_3x3)

        # 5x5 convolution branch
        x_5x5 = nn.Conv(self.c_red["5x5"], kernel_size=(1, 1), kernel_init=googlenet_
        ↵ kernel_init, use_bias=False)(x)
```

(continues on next page)

(continued from previous page)

```

x_5x5 = nn.BatchNorm()(x_5x5, use_running_average=not train)
x_5x5 = self.act_fn(x_5x5)
x_5x5 = nn.Conv(self.c_out["5x5"], kernel_size=(5, 5), kernel_init=googlenet_
kernel_init, use_bias=False)(x_5x5)
x_5x5 = nn.BatchNorm()(x_5x5, use_running_average=not train)
x_5x5 = self.act_fn(x_5x5)

# Max-pool branch
x_max = nn.max_pool(x, (3, 3), strides=(2, 2))
x_max = nn.Conv(self.c_out["max"], kernel_size=(1, 1), kernel_init=googlenet_
kernel_init, use_bias=False)(x)
x_max = nn.BatchNorm()(x_max, use_running_average=not train)
x_max = self.act_fn(x_max)

x_out = jnp.concatenate([x_1x1, x_3x3, x_5x5, x_max], axis=-1)
return x_out

```

The GoogleNet architecture consists of stacking multiple Inception blocks with occasional max pooling to reduce the height and width of the feature maps. The original GoogleNet was designed for image sizes of ImageNet (224x224 pixels) and had almost 7 million parameters. As we train on CIFAR10 with image sizes of 32x32, we don't require such a heavy architecture, and instead, apply a reduced version. The number of channels for dimensionality reduction and output per filter (1x1, 3x3, 5x5, and max pooling) need to be manually specified and can be changed if interested. The general intuition is to have the most filters for the 3x3 convolutions, as they are powerful enough to take the context into account while requiring almost a third of the parameters of the 5x5 convolution.

```
[12]: class GoogleNet(nn.Module):
    num_classes : int
    act_fn : callable

    @nn.compact
    def __call__(self, x, train=True):
        # A first convolution on the original image to scale up the channel size
        x = nn.Conv(64, kernel_size=(3, 3), kernel_init=googlenet_kernel_init, use_
        bias=False)(x)
        x = nn.BatchNorm()(x, use_running_average=not train)
        x = self.act_fn(x)

        # Stacking inception blocks
        inception_blocks = [
            InceptionBlock(c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 16, "3x3": 32,
            "5x5": 8, "max": 8}, act_fn=self.act_fn),
            InceptionBlock(c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 24, "3x3": 48,
            "5x5": 12, "max": 12}, act_fn=self.act_fn),
            lambda inp: nn.max_pool(inp, (3, 3), strides=(2, 2)), # 32x32 => 16x16
            InceptionBlock(c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 24, "3x3": 48,
            "5x5": 12, "max": 12}, act_fn=self.act_fn),
            InceptionBlock(c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 16, "3x3": 48,
            "5x5": 16, "max": 16}, act_fn=self.act_fn),
            InceptionBlock(c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 16, "3x3": 48,
            "5x5": 16, "max": 16}, act_fn=self.act_fn),
            InceptionBlock(c_red={"3x3": 32, "5x5": 16}, c_out={"1x1": 32, "3x3": 48,
            "5x5": 24, "max": 24}, act_fn=self.act_fn),
        ]

```

(continues on next page)

(continued from previous page)

```

lambda inp: nn.max_pool(inp, (3, 3), strides=(2, 2)), # 16x16 => 8x8
    InceptionBlock(c_red={"3x3": 48, "5x5": 16}, c_out={"1x1": 32, "3x3": 64,
    ↵"5x5": 16, "max": 16}, act_fn=self.act_fn),
        InceptionBlock(c_red={"3x3": 48, "5x5": 16}, c_out={"1x1": 32, "3x3": 64,
    ↵"5x5": 16, "max": 16}, act_fn=self.act_fn)
    ]
    for block in inception_blocks:
        x = block(x, train=train) if isinstance(block, InceptionBlock) else block(x)

        # Mapping to classification output
        x = x.mean(axis=(1, 2))
        x = nn.Dense(self.num_classes)(x)
    return x

```

The training of the model is handled by our previously implemented training function, and we just have to define the command to start. Note that we train for 200 epochs, which takes about 15mins on a RTX3090. We would recommend using the saved models and train your own model if you are interested.

```
[13]: googlenet_trainer, googlenet_results = train_classifier(model_class=GoogleNet,
                                                               model_name="GoogleNet",
                                                               model_hparams={"num_classes": 10,
                                                               "act_fn": nn.relu}

    ↵,
                                                               optimizer_name="adamm",
                                                               optimizer_hparams={"lr": 1e-3,
                                                               "weight_decay
    ↵": 1e-4},
                                                               exmp_imgs=jax.device_put(
                                                               next(iterator(train_loader))[0]),
                                                               num_epochs=200)
```

We will compare the results later in the notebooks, but we can already print them here for a first glance:

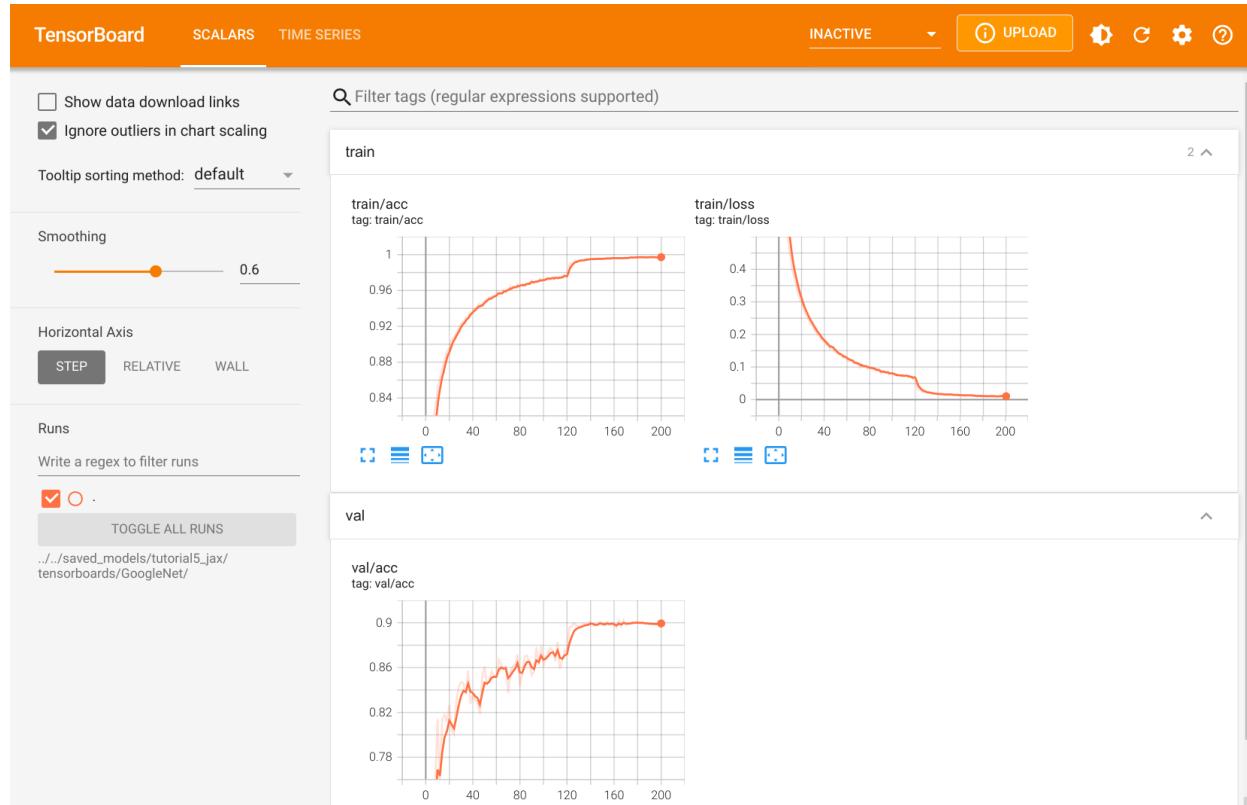
```
[14]: print("GoogleNet Results", googlenet_results)
GoogleNet Results {'val': 0.9022000432014465, 'test': 0.8911000490188599}
```

Tensorboard log

For logging our results, we have used TensorBoard. To give you a better intuition of what TensorBoard can be used for, we can look at the board that our training of the GoogleNet has generated. TensorBoard provides an inline functionality for Jupyter notebooks, and we use it here:

```
[15]: # Load tensorboard extension
%load_ext tensorboard
```

```
[16]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH!
%tensorboard --logdir ../../saved_models/tutorial5_jax/tensorboards/GoogleNet/
```



TensorBoard is organized in multiple tabs. The main tab is the scalar tab where we can log the development of single numbers. For example, we have plotted the training loss, accuracy, learning rate, etc. If we look at the training or validation accuracy, we can really see the impact of using a learning rate scheduler. Reducing the learning rate gives our model a nice increase in training performance. Similarly, when looking at the training loss, we see a sudden decrease at this point. However, the high numbers on the training set compared to validation indicate that our model was overfitting which is inevitable for such large networks.

4.22.3 ResNet

The ResNet paper is one of the [most cited AI papers](#), and has been the foundation for neural networks with more than 1,000 layers. Despite its simplicity, the idea of residual connections is highly effective as it supports stable gradient propagation through the network. Instead of modeling $x_{l+1} = F(x_l)$, we model $x_{l+1} = x_l + F(x_l)$ where F is a non-linear mapping (usually a sequence of NN modules like convolutions, activation functions, and normalizations). If we do backpropagation on such residual connections, we obtain:

$$\frac{\partial x_{l+1}}{\partial x_l} = \mathbf{I} + \frac{\partial F(x_l)}{\partial x_l}$$

The bias towards the identity matrix guarantees a stable gradient propagation being less effected by F itself. There have been many variants of ResNet proposed, which mostly concern the function F , or operations applied on the sum. In this tutorial, we look at two of them: the original ResNet block, and the [Pre-Activation ResNet block](#). We visually compare the blocks below (figure credit - [He et al.](#)):

The original ResNet block applies a non-linear activation function, usually ReLU, after the skip connection. In contrast, the pre-activation ResNet block applies the non-linearity at the beginning of F . Both have their advantages and disadvantages. For very deep network, however, the pre-activation ResNet has shown to perform better as the gradient flow

is guaranteed to have the identity matrix as calculated above, and is not harmed by any non-linear activation applied to it. For comparison, in this notebook, we implement both ResNet types as shallow networks.

Let's start with the original ResNet block. The visualization above already shows what layers are included in F . One special case we have to handle is when we want to reduce the image dimensions in terms of width and height. The basic ResNet block requires $F(x_l)$ to be of the same shape as x_l . Thus, we need to change the dimensionality of x_l as well before adding to $F(x_l)$. The original implementation used an identity mapping with stride 2 and padded additional feature dimensions with 0. However, the more common implementation is to use a 1x1 convolution with stride 2 as it allows us to change the feature dimensionality while being efficient in parameter and computation cost. The code for the ResNet block is relatively simple, and shown below:

```
[17]: # Conv initialized with kaiming int, but uses fan-out instead of fan-in mode
# Fan-out focuses on the gradient distribution, and is commonly used in ResNets
resnet_kernel_init = nn.initializers.variance_scaling(2.0, mode='fan_out', distribution=
    ↪'normal')

class ResNetBlock(nn.Module):
    act_fn : callable # Activation function
    c_out : int # Output feature size
    subsample : bool = False # If True, we apply a stride inside F

    @nn.compact
    def __call__(self, x, train=True):
        # Network representing F
        z = nn.Conv(self.c_out, kernel_size=(3, 3),
                   strides=(1, 1) if not self.subsample else (2, 2),
                   kernel_init=resnet_kernel_init,
                   use_bias=False)(x)
        z = nn.BatchNorm()(z, use_running_average=not train)
        z = self.act_fn(z)
        z = nn.Conv(self.c_out, kernel_size=(3, 3),
                   kernel_init=resnet_kernel_init,
                   use_bias=False)(z)
        z = nn.BatchNorm()(z, use_running_average=not train)

        if self.subsample:
            x = nn.Conv(self.c_out, kernel_size=(1, 1), strides=(2, 2), kernel_
    ↪init=resnet_kernel_init)(x)

        x_out = self.act_fn(z + x)
        return x_out
```

The second block we implement is the pre-activation ResNet block. For this, we have to change the order of layers, and do not apply an activation function on the output. Additionally, the downsampling operation has to apply a non-linearity as well as the input, x_l , has not been processed by a non-linearity yet. Hence, the block looks as follows:

```
[18]: class PreActResNetBlock(ResNetBlock):

    @nn.compact
    def __call__(self, x, train=True):
        # Network representing F
        z = nn.BatchNorm()(x, use_running_average=not train)
        z = self.act_fn(z)
        z = nn.Conv(self.c_out, kernel_size=(3, 3),
```

(continues on next page)

(continued from previous page)

```

        strides=(1, 1) if not self.subsample else (2, 2),
        kernel_init=resnet_kernel_init,
        use_bias=False)(z)
    z = nn.BatchNorm()(z, use_running_average=not train)
    z = self.act_fn(z)
    z = nn.Conv(self.c_out, kernel_size=(3, 3),
               kernel_init=resnet_kernel_init,
               use_bias=False)(z)

    if self.subsample:
        x = nn.BatchNorm()(x, use_running_average=not train)
        x = self.act_fn(x)
        x = nn.Conv(self.c_out,
                   kernel_size=(1, 1),
                   strides=(2, 2),
                   kernel_init=resnet_kernel_init,
                   use_bias=False)(x)

    x_out = z + x
    return x_out

```

The overall ResNet architecture consists of stacking multiple ResNet blocks, of which some are downsampling the input. When talking about ResNet blocks in the whole network, we usually group them by the same output shape. Hence, if we say the ResNet has [3, 3, 3] blocks, it means that we have 3 times a group of 3 ResNet blocks, where a subsampling is taking place in the fourth and seventh block. The ResNet with [3, 3, 3] blocks on CIFAR10 is visualized below.

The three groups operate on the resolutions 32×32 , 16×16 and 8×8 respectively. The blocks in orange denote ResNet blocks with downsampling. The same notation is used by many other implementations such as in the `torchvision` library from PyTorch or `flaxmodels` (pretrained ResNets and more for JAX). Thus, our code looks as follows:

```
[19]: class ResNet(nn.Module):
    num_classes : int
    act_fn : callable
    block_class : nn.Module
    num_blocks : tuple = (3, 3, 3)
    c_hidden : tuple = (16, 32, 64)

    @nn.compact
    def __call__(self, x, train=True):
        # A first convolution on the original image to scale up the channel size
        x = nn.Conv(self.c_hidden[0], kernel_size=(3, 3), kernel_init=resnet_kernel_init,
        ↵ use_bias=False)(x)
        if self.block_class == ResNetBlock: # If pre-activation block, we do not apply ↵
        ↵ non-linearities yet
            x = nn.BatchNorm()(x, use_running_average=not train)
            x = self.act_fn(x)

        # Creating the ResNet blocks
        for block_idx, block_count in enumerate(self.num_blocks):
            for bc in range(block_count):
                # Subsample the first block of each group, except the very first one.

```

(continues on next page)

(continued from previous page)

```

subsample = (bc == 0 and block_idx > 0)
# ResNet block
x = self.block_class(c_out=self.c_hidden[block_idx],
                      act_fn=self.act_fn,
                      subsample=subsample)(x, train=train)

# Mapping to classification output
x = x.mean(axis=(1, 2))
x = nn.Dense(self.num_classes)(x)
return x

```

Finally, we can train our ResNet models. One difference to the GoogleNet training is that we explicitly use SGD with Momentum as optimizer instead of Adam. Adam often leads to a slightly worse accuracy on plain, shallow ResNets. It is not 100% clear why Adam performs worse in this context, but one possible explanation is related to ResNet's loss surface. ResNet has been shown to produce smoother loss surfaces than networks without skip connection (see Li et al., 2018 for details). A possible visualization of the loss surface with/out skip connections is below (figure credit - Li et al.):

The x and y axis shows a projection of the parameter space, and the z axis shows the loss values achieved by different parameter values. On smooth surfaces like the one on the right, we might not require an adaptive learning rate as Adam provides. Instead, Adam can get stuck in local optima while SGD finds the wider minima that tend to generalize better. However, to answer this question in detail, we would need an extra tutorial because it is not easy to answer. For now, we conclude: for ResNet architectures, consider the optimizer to be an important hyperparameter, and try training with both Adam and SGD. Let's train the model below with SGD:

```
[20]: resnet_trainer, resnet_results = train_classifier(model_name="ResNet",
                                                       model_class=ResNet,
                                                       model_hparams={"num_classes": 10,
                                                                     "c_hidden": (16, 32, 64),
                                                                     "num_blocks": (3, 3, 3),
                                                                     "act_fn": nn.relu,
                                                                     "block_class": ResNetBlock},
                                                       optimizer_name="SGD",
                                                       optimizer_hparams={"lr": 0.1,
                                                                     "momentum": 0.9,
                                                                     "weight_decay": 1e-4},
                                                       exmp_imgs=jax.device_put(next(iter(train_loader))[0]),
                                                       num_epochs=200)
```

Let's also train the pre-activation ResNet as comparison:

```
[21]: preactresnet_trainer, preactresnet_results = train_classifier(model_name="PreActResNet",
                                                               model_class=ResNet,
                                                               model_hparams={"num_classes": 10,
                                                                             "c_hidden": (16, 32, 64),
```

(continues on next page)

(continued from previous page)

```

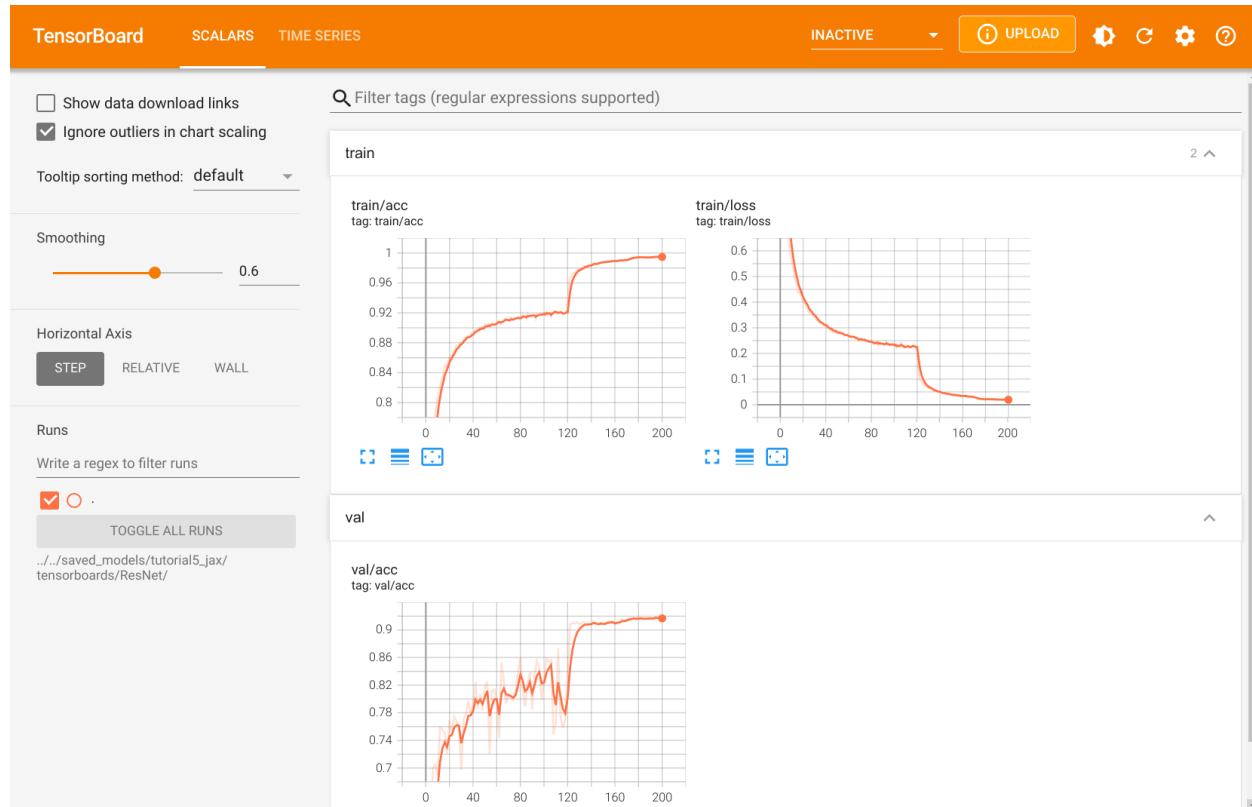
"num_blocks": 3,
"act_fn": "relu",
"block_class": "PreActResNetBlock",
"optimizer_name": "SGD",
"optimizer_hparams": {"lr": 0.1,
"momentum": 0.9,
"decay": 1e-4},
"weight_exmp_imgs=jax.device_put(
next(iterator(train_loader))[0]),
num_epochs=200)

```

Tensorboard log

Similarly to our GoogleNet model, we also have a TensorBoard log for the ResNet model. We can open it below.

```
[22]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH! Feel free to
      # change "ResNet" to "PreActResNet"
%tensorboard --logdir ../../saved_models/tutorial5_jax/tensorboards/ResNet/
```



Feel free to explore the TensorBoard yourself, including the computation graph. In general, we can see that with SGD, the ResNet has a higher training loss than the GoogleNet in the first stage of the training. After reducing the learning rate however, the model achieves even higher validation accuracies. We compare the precise scores at the end of the notebook.

4.22.4 DenseNet

DenseNet is another architecture for enabling very deep neural networks and takes a slightly different perspective on residual connections. Instead of modeling the difference between layers, DenseNet considers residual connections as a possible way to reuse features across layers, removing any necessity to learn redundant feature maps. If we go deeper into the network, the model learns abstract features to recognize patterns. However, some complex patterns consist of a combination of abstract features (e.g. hand, face, etc.), and low-level features (e.g. edges, basic color, etc.). To find these low-level features in the deep layers, standard CNNs have to learn copy such feature maps, which wastes a lot of parameter complexity. DenseNet provides an efficient way of reusing features by having each convolution depends on all previous input features, but add only a small amount of filters to it. See the figure below for an illustration (figure credit - [Hu et al.](#)):

The last layer, called the transition layer, is responsible for reducing the dimensionality of the feature maps in height, width, and channel size. Although those technically break the identity backpropagation, there are only a few in a network so that it doesn't affect the gradient flow much.

We split the implementation of the layers in DenseNet into three parts: a `DenseLayer`, and a `DenseBlock`, and a `TransitionLayer`. The module `DenseLayer` implements a single layer inside a dense block. It applies a 1x1 convolution for dimensionality reduction with a subsequent 3x3 convolution. The output channels are concatenated to the originals and returned. Note that we apply the Batch Normalization as the first layer of each block. This allows slightly different activations for the same features to different layers, depending on what is needed. Overall, we can implement it as follows:

```
[23]: densenet_kernel_init = nn.initializers.kaiming_normal()

class DenseLayer(nn.Module):
    bn_size : int # Bottleneck size (factor of growth rate) for the output of the 1x1 convolution
    growth_rate : int # Number of output channels of the 3x3 convolution
    act_fn : callable # Activation function

    @nn.compact
    def __call__(self, x, train=True):
        z = nn.BatchNorm()(x, use_running_average=not train)
        z = self.act_fn(z)
        z = nn.Conv(self.bn_size * self.growth_rate,
                   kernel_size=(1, 1),
                   kernel_init=densenet_kernel_init,
                   use_bias=False)(z)
        z = nn.BatchNorm()(z, use_running_average=not train)
        z = self.act_fn(z)
        z = nn.Conv(self.growth_rate,
                   kernel_size=(3, 3),
                   kernel_init=densenet_kernel_init,
                   use_bias=False)(z)
        x_out = jnp.concatenate([x, z], axis=-1)
        return x_out
```

The module `DenseBlock` summarizes multiple dense layers applied in sequence. Each dense layer takes as input the original input concatenated with all previous layers' feature maps:

```
[24]: class DenseBlock(nn.Module):
    num_layers : int # Number of dense layers to apply in the block
    bn_size : int # Bottleneck size to use in the dense layers
    growth_rate : int # Growth rate to use in the dense layers
    act_fn : callable # Activation function to use in the dense layers

    @nn.compact
    def __call__(self, x, train=True):
        for _ in range(self.num_layers):
            x = DenseLayer(bn_size=self.bn_size,
                           growth_rate=self.growth_rate,
                           act_fn=self.act_fn)(x, train=train)
    return x
```

Finally, the `TransitionLayer` takes as input the final output of a dense block and reduces its channel dimensionality using a 1x1 convolution. To reduce the height and width dimension, we take a slightly different approach than in ResNet and apply an average pooling with kernel size 2 and stride 2. This is because we don't have an additional connection to the output that would consider the full 2x2 patch instead of a single value. Besides, it is more parameter efficient than using a 3x3 convolution with stride 2. Thus, the layer is implemented as follows:

```
[25]: class TransitionLayer(nn.Module):
    c_out : int # Output feature size
    act_fn : callable # Activation function

    @nn.compact
    def __call__(self, x, train=True):
        x = nn.BatchNorm()(x, use_running_average=not train)
        x = self.act_fn(x)
        x = nn.Conv(self.c_out,
                   kernel_size=(1, 1),
                   kernel_init=densenet_kernel_init,
                   use_bias=False)(x)
        x = nn.avg_pool(x, (2, 2), strides=(2, 2))
    return x
```

Now we can put everything together and create our DenseNet. To specify the number of layers, we use a similar notation as in ResNets and pass on a list of ints representing the number of layers per block. After each dense block except the last one, we apply a transition layer to reduce the dimensionality by 2.

```
[26]: class DenseNet(nn.Module):
    num_classes : int
    act_fn : callable = nn.relu
    num_layers : tuple = (6, 6, 6, 6)
    bn_size : int = 2
    growth_rate : int = 16

    @nn.compact
    def __call__(self, x, train=True):
        c_hidden = self.growth_rate * self.bn_size # The start number of hidden channels

        x = nn.Conv(c_hidden,
```

(continues on next page)

(continued from previous page)

```

        kernel_size=(3, 3),
        kernel_init=densenet_kernel_init)(x)

    for block_idx, num_layers in enumerate(self.num_layers):
        x = DenseBlock(num_layers=num_layers,
                       bn_size=self.bn_size,
                       growth_rate=self.growth_rate,
                       act_fn=self.act_fn)(x, train=train)
        c_hidden += num_layers * self.growth_rate
        if block_idx < len(self.num_layers)-1: # Don't apply transition layer on
            ↵last block
            x = TransitionLayer(c_out=c_hidden//2,
                                 act_fn=self.act_fn)(x, train=train)
            c_hidden //= 2

        x = nn.BatchNorm()(x, use_running_average=not train)
        x = self.act_fn(x)
        x = x.mean(axis=(1, 2))
        x = nn.Dense(self.num_classes)(x)
    return x

```

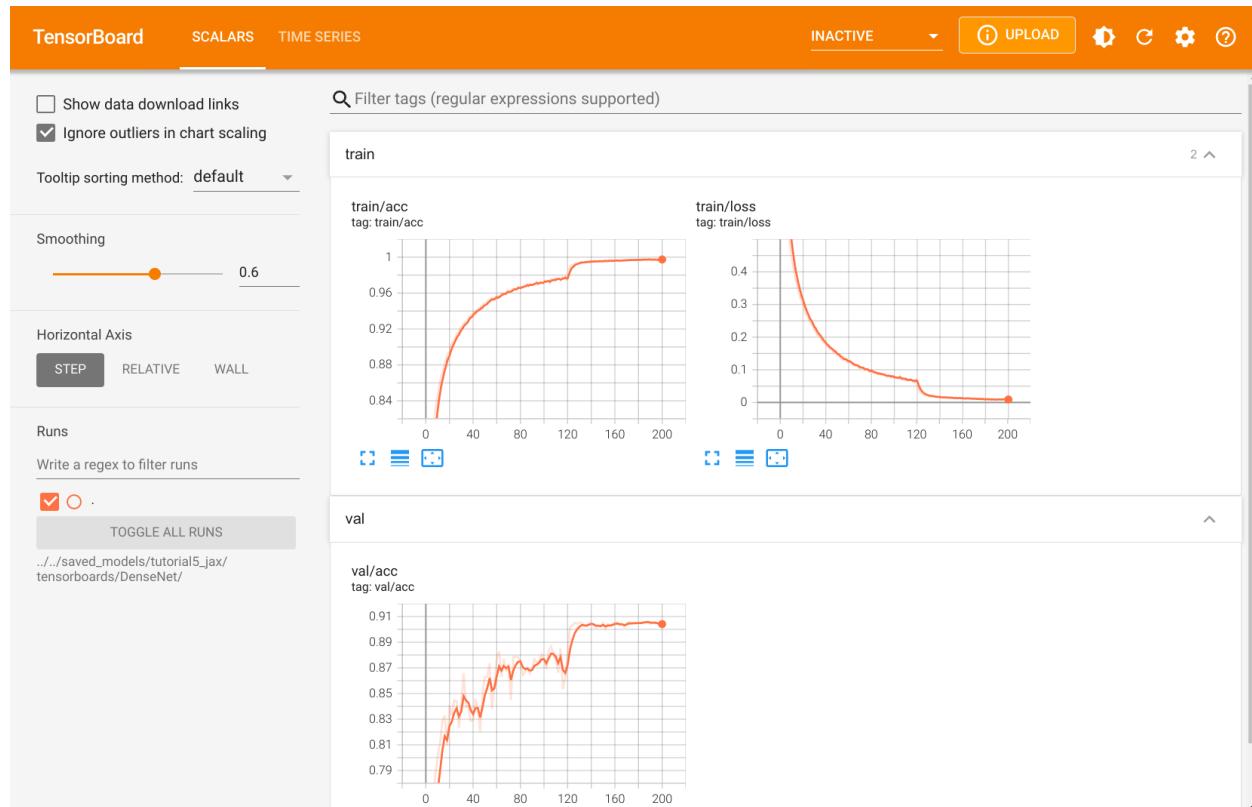
Lastly, we train our network. In contrast to ResNet, DenseNet does not show any issues with Adam, and hence we train it with this optimizer. The other hyperparameters are chosen to result in a network with a similar parameter size as the ResNet and GoogleNet. Commonly, when designing very deep networks, DenseNet is more parameter efficient than ResNet while achieving a similar or even better performance.

```
[27]: densenet_trainer, densenet_results = train_classifier(model_name="DenseNet",
                                                       model_class=DenseNet,
                                                       model_hparams={"num_classes": 10,
                                                                     "num_layers": [6, 6,
                                                                       ↵ 6, 6],
                                                                     "bn_size": 2,
                                                                     "growth_rate": 16,
                                                                     "act_fn": nn.relu},
                                                       optimizer_name="adamw",
                                                       optimizer_hparams={"lr": 1e-3,
                                                                     "weight_decay": 1e-4},
                                                       exmp_imgs=jax.device_put(
                                                       next(iter(train_loader))[0]),
                                                       num_epochs=200)
```

Tensorboard log

Finally, we also have another TensorBoard for the DenseNet training. We take a look at it below:

```
[28]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH!
%tensorboard --logdir ../../saved_models/tutorials5_jax/tensorboards/DenseNet/
```



The overall course of the validation accuracy and training loss resemble the training of GoogleNet, which is also related to training the network with Adam. Feel free to explore the training metrics yourself.

4.22.5 Conclusion and Comparison

After discussing each model separately, and training all of them, we can finally compare them. First, let's organize the results of all models in a table:

```
[29]: %%html
<!-- Some HTML code to increase font size in the following table -->
<style>
th {font-size: 120%;}
td {font-size: 120%;}
</style>
<IPython.core.display.HTML object>
```

```
[30]: import tabulate
from IPython.display import display, HTML
all_models = [
```

(continues on next page)

(continued from previous page)

```

("GoogleNet", googlenet_results, googlenet_trainer),
("ResNet", resnet_results, resnet_trainer),
("PreActResNet", preactresnet_results, preactresnet_trainer),
("DenseNet", densenet_results, densenet_trainer)
]
table = [[model_name,
          f"{100.0*model_results['val']:.2f}%",
          f"{100.0*model_results['test']:.2f}%",
          "{:,}".format(sum([np.prod(p.shape) for p in jax.tree_leaves(model.state.
          ↪params)]))]
           for model_name, model_results, model in all_models]
display(HTML(tabulate.tabulate(table, tablefmt='html', headers=["Model", "Val Accuracy",
          ↪"Test Accuracy", "Num Parameters"])))
<IPython.core.display.HTML object>

```

First of all, we see that all models are performing reasonably well. Simple models as you have implemented them in the practical achieve considerably lower performance, which is beside the lower number of parameters also attributed to the architecture design choice. GoogleNet is the model to obtain the lowest performance on the validation and test set, although it is very close to DenseNet. A proper hyperparameter search over all the channel sizes in GoogleNet would likely improve the accuracy of the model to a similar level, but this is also expensive given a large number of hyperparameters. ResNet outperforms both DenseNet and GoogleNet by more than 1% on the validation set, while there is a minor difference between both versions, original and pre-activation. We can conclude that for shallow networks, the place of the activation function does not seem to be crucial, although papers have reported the contrary for very deep networks (e.g. He et al.).

In general, we can conclude that ResNet is a simple, but powerful architecture. If we would apply the models on more complex tasks with larger images and more layers inside the networks, we would likely see a bigger gap between GoogleNet and skip-connection architectures like ResNet and DenseNet. A comparison with deeper models on CIFAR10 can be for example found [here](#). Interestingly, DenseNet outperforms the original ResNet on their setup but comes closely behind the Pre-Activation ResNet. The best model, a Dual Path Network (Chen et. al), is actually a combination of ResNet and DenseNet showing that both offer different advantages.

Which model should I choose for my task?

We have reviewed four different models. So, which one should we choose if have given a new task? Usually, starting with a ResNet is a good idea given the superior performance of the CIFAR dataset and its simple implementation. Besides, for the parameter number we have chosen here, ResNet is the fastest as DenseNet and GoogleNet have many more layers that are applied in sequence in our primitive implementation. However, if you have a really difficult task, such as semantic segmentation on HD images, more complex variants of ResNet and DenseNet are recommended.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.23 Tutorial 6 (JAX): Transformers and Multi-Head Attention

Filled notebook:

Pre-trained models:

PyTorch version:

Author: Phillip Lippe

Note: This notebook is written in JAX+Flax. It is a 1-to-1 translation of the original notebook written in PyTorch+PyTorch Lightning with almost identical results. For an introduction to JAX, check out our [Tutorial 2 \(JAX\): Introduction to JAX+Flax](#). Further, throughout the notebook, we comment on major differences to the PyTorch version and provide explanations for the major parts of the JAX code.

Speed comparison: We note the training times for all models in the PyTorch and the JAX implementation below (PyTorch v1.11, JAX v0.3.13). The models were trained on the same hardware (NVIDIA RTX3090, 24 core CPU) and we slightly adjusted the tutorials to use the exact same training settings (same data loading parameters, evaluation schedule, etc.). Overall, the JAX implementation is almost *4x faster* than PyTorch! However, this is mostly due to the small model and input sizes, and the code has not been explicitly designed for benchmarking. With larger models, larger batch sizes, or smaller GPUs, the speed up is expected to become considerably smaller (see e.g. [Tutorial 15](#)).

Models	PyTorch	JAX
Reverse Sequence	0min 26sec	0min 7sec
Anomaly Detection	16min 34sec	3min 45sec

In this tutorial, we will discuss one of the most impactful architectures of the last 2 years: the Transformer model. Since the paper [Attention Is All You Need](#) by Vaswani et al. had been published in 2017, the Transformer architecture has continued to beat benchmarks in many domains, most importantly in Natural Language Processing. Transformers with an incredible amount of parameters can generate long, convincing [essays](#), and opened up new application fields of AI. As the hype of the Transformer architecture seems not to come to an end in the next years, it is important to understand how it works, and have implemented it yourself, which we will do in this notebook.

Despite the huge success of Transformers in NLP, we will *not* include the NLP domain in our notebook here. Why? Firstly, the Master AI at UvA offers many great NLP courses that will take a closer look at the application of the Transformer architecture in NLP ([NLP2](#), [Advanced Topics in Computational Semantics](#)). Secondly, assignment 2 takes already a closer look at language generation on character level, on which you could easily apply our transformer architecture. Finally, and most importantly, there is so much more to the Transformer architecture. NLP is the domain the Transformer architecture has been originally proposed for and had the greatest impact on, but it also accelerated research in other domains, recently even [Computer Vision](#). Thus, we focus here on what makes the Transformer and self-attention so powerful in general. In [Tutorial 15](#), we will discuss the application of Transformers in Computer Vision.

Below, we import our standard libraries. We use [JAX](#) as acceleration backend, [Flax](#) for implementing neural networks, and [Optax](#) to optimize the models.

```
[1]: ## Standard libraries
import os
import numpy as np
import math
import json
from functools import partial

## Imports for plotting
import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## tqdm for loading bars
from tqdm.auto import tqdm

## To run JAX on TPU in Google Colab, uncomment the two lines below
# import jax.tools.colab_tpu
# jax.tools.colab_tpu.setup_tpu()

## JAX
import jax
import jax.numpy as jnp
from jax import random

## Flax (NN in JAX)
try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax
from flax import linen as nn
from flax.training import train_state, checkpoints

## Optax (Optimizers in JAX)
try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax

## PyTorch
import torch
import torch.utils.data as data
from torch.utils.tensorboard import SummaryWriter
import torchvision
from torchvision import transforms
```

(continues on next page)

(continued from previous page)

```
from torchvision.datasets import CIFAR100

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../../saved_models/tutorial6_jax"

# Seeding for random operations
main_rng = random.PRNGKey(42)

print("Device:", jax.devices()[0])
Device: gpu:0
```

Two pre-trained models are downloaded below. Make sure to have adjusted your CHECKPOINT_PATH before running this code if not already done.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/JAX/tutorial6/"
# Files to download
pretrained_files = ["ReverseTask.ckpt", "SetAnomalyTask.ckpt"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive folder, or contact the author with the full output including the following error:\n", e)
```

4.23.1 The Transformer architecture

In the first part of this notebook, we will implement the Transformer architecture by hand. As the architecture is so popular, its main components are already integrated into Flax (SelfAttention, MultiHeadAttention) and there exist several implementations (e.g. in Trax) and pre-trained models (e.g. on Hugging Face). However, we will implement it here ourselves, to get through to the smallest details.

There are of course many more tutorials out there about attention and Transformers. Below, we list a few that are worth exploring if you are interested in the topic and might want yet another perspective on the topic after this one:

- Transformer: A Novel Neural Network Architecture for Language Understanding (Jakob Uszkoreit, 2017) - The original Google blog post about the Transformer paper, focusing on the application in machine translation.
- The Illustrated Transformer (Jay Alammar, 2018) - A very popular and great blog post intuitively explaining the Transformer architecture with many nice visualizations. The focus is on NLP.
- Attention? Attention! (Lilian Weng, 2018) - A nice blog post summarizing attention mechanisms in many domains including vision.
- Illustrated: Self-Attention (Raimi Karim, 2019) - A nice visualization of the steps of self-attention. Recommended going through if the explanation below is too abstract for you.
- The Transformer family (Lilian Weng, 2020) - A very detailed blog post reviewing more variants of Transformers besides the original one.

What is Attention?

The attention mechanism describes a recent new group of layers in neural networks that has attracted a lot of interest in the past few years, especially in sequence tasks. There are a lot of different possible definitions of “attention” in the literature, but the one we will use here is the following: *the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements’ keys*. So what does this exactly mean? The goal is to take an average over the features of multiple elements. However, instead of weighting each element equally, we want to weight them depending on their actual values. In other words, we want to dynamically decide on which inputs we want to “attend” more than others. In particular, an attention mechanism has usually four parts we need to specify:

- **Query:** The query is a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.
- **Keys:** For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is “offering”, or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.
- **Values:** For each input element, we also have a value vector. This feature vector is the one we want to average over.
- **Score function:** To rate which elements we want to pay attention to, we need to specify a score function f_{attn} . The score function takes the query and a key as input, and output the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product, or a small MLP.

The weights of the average are calculated by a softmax over all score function outputs. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query. If we try to describe it with pseudo-math, we can write:

$$\alpha_i = \frac{\exp(f_{attn}(\text{key}_i, \text{query}))}{\sum_j \exp(f_{attn}(\text{key}_j, \text{query}))}, \quad \text{out} = \sum_i \alpha_i \cdot \text{value}_i$$

Visually, we can show the attention over a sequence of words as follows:

For every word, we have one key and one value vector. The query is compared to all keys with a score function (in this case the dot product) to determine the weights. The softmax is not visualized for simplicity. Finally, the value vectors of all words are averaged using the attention weights.

Most attention mechanisms differ in terms of what queries they use, how the key and value vectors are defined, and what score function is used. The attention applied inside the Transformer architecture is called **self-attention**. In self-attention, each sequence element provides a key, value, and query. For each element, we perform an attention layer where based on its query, we check the similarity of the all sequence elements’ keys, and returned a different, averaged

value vector for each element. We will now go into a bit more detail by first looking at the specific implementation of the attention mechanism which is in the Transformer case the scaled dot product attention.

Scaled Dot Product Attention

The core concept behind self-attention is the scaled dot product attention. Our goal is to have an attention mechanism with which any element in a sequence can attend to any other while still being efficient to compute. The dot product attention takes as input a set of queries $Q \in \mathbb{R}^{T \times d_k}$, keys $K \in \mathbb{R}^{T \times d_k}$ and values $V \in \mathbb{R}^{T \times d_v}$ where T is the sequence length, and d_k and d_v are the hidden dimensionality for queries/keys and values respectively. For simplicity, we neglect the batch dimension for now. The attention value from element i to j is based on its similarity of the query Q_i and key K_j , using the dot product as the similarity metric. In math, we calculate the dot product attention as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The matrix multiplication QK^T performs the dot product for every possible pair of queries and keys, resulting in a matrix of the shape $T \times T$. Each row represents the attention logits for a specific element i to all other elements in the sequence. On these, we apply a softmax and multiply with the value vector to obtain a weighted mean (the weights being determined by the attention). Another perspective on this attention mechanism offers the computation graph which is visualized below (figure credit - [Vaswani et al., 2017](#)).

One aspect we haven't discussed yet is the scaling factor of $1/\sqrt{d_k}$. This scaling factor is crucial to maintain an appropriate variance of attention values after initialization. Remember that we initialize our layers with the intention of having equal variance throughout the model, and hence, Q and K might also have a variance close to 1. However, performing a dot product over two vectors with a variance σ^2 results in a scalar having d_k -times higher variance:

$$q_i \sim \mathcal{N}(0, \sigma^2), k_i \sim \mathcal{N}(0, \sigma^2) \rightarrow \text{Var}\left(\sum_{i=1}^{d_k} q_i \cdot k_i\right) = \sigma^4 \cdot d_k$$

If we do not scale down the variance back to $\sim \sigma^2$, the softmax over the logits will already saturate to 1 for one random element and 0 for all others. The gradients through the softmax will be close to zero so that we can't learn the parameters appropriately. Note that the extra factor of σ^2 , i.e., having σ^4 instead of σ^2 , is usually not an issue, since we keep the original variance σ^2 close to 1 anyways.

The block `Mask (opt.)` in the diagram above represents the optional masking of specific entries in the attention matrix. This is for instance used if we stack multiple sequences with different lengths into a batch. To still benefit from parallelization in PyTorch, we pad the sentences to the same length and mask out the padding tokens during the calculation of the attention values. This is usually done by setting the respective attention logits to a very low value.

After we have discussed the details of the scaled dot product attention block, we can write a function below which computes the output features given the triple of queries, keys, and values:

```
[3]: def scaled_dot_product(q, k, v, mask=None):
    d_k = q.shape[-1]
    attn_logits = jnp.matmul(q, jnp.swapaxes(k, -2, -1))
    attn_logits = attn_logits / math.sqrt(d_k)
    if mask is not None:
        attn_logits = jnp.where(mask == 0, -9e15, attn_logits)
    attention = nn.softmax(attn_logits, axis=-1)
    values = jnp.matmul(attention, v)
    return values, attention
```

Note that our code above supports any additional dimensionality in front of the sequence length so that we can also use it for batches. However, for a better understanding, let's generate a few random queries, keys, and value vectors, and calculate the attention outputs:

```
[4]: seq_len, d_k = 3, 2
main_rng, rand1 = random.split(main_rng)
qkv = random.normal(rand1, (3, seq_len, d_k))
q, k, v = qkv[0], qkv[1], qkv[2]
values, attention = scaled_dot_product(q, k, v)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Values\n", values)
print("Attention\n", attention)

Q
[[[-0.6613315  0.70056266]
 [ 0.08239268 -1.7793142 ]
 [-0.04378588  1.0965251 ]]

K
[[ 1.7257481  0.35568172]
 [ 1.3034704  1.2873708 ]
 [ 1.6871481 -0.5714404 ]]

V
[[ 1.5129997  1.1050899 ]
 [ 0.27949408 -0.46224892]
 [-1.1003422 -1.1437942 ]]

Values
[[ 0.376226  -0.14656176]
 [-0.42778552 -0.5989564 ]
 [ 0.4362476 -0.11678296]]

Attention
[[0.27963293 0.54049295 0.17987415]
 [0.22194655 0.06706189 0.71099156]
 [0.27977085 0.58373076 0.13649833]]
```

Before continuing, make sure you can follow the calculation of the specific values here, and also check it by hand. It is important to fully understand how the scaled dot product attention is calculated.

Multi-Head Attention

The scaled dot product attention allows a network to attend over a sequence. However, often there are multiple different aspects a sequence element wants to attend to, and a single weighted average is not a good option for it. This is why we extend the attention mechanisms to multiple heads, i.e. multiple different query-key-value triplets on the same features. Specifically, given a query, key, and value matrix, we transform those into h sub-queries, sub-keys, and sub-values, which we pass through the scaled dot product attention independently. Afterward, we concatenate the heads and combine them with a final weight matrix. Mathematically, we can express this operation as:

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

We refer to this as Multi-Head Attention layer with the learnable parameters $W_{1\dots h}^Q \in \mathbb{R}^{D \times d_k}$, $W_{1\dots h}^K \in \mathbb{R}^{D \times d_k}$, $W_{1\dots h}^V \in \mathbb{R}^{D \times d_v}$, and $W^O \in \mathbb{R}^{h \cdot d_k \times d_{out}}$ (D being the input dimensionality). Expressed in a computational graph, we can visualize it as below (figure credit - [Vaswani et al., 2017](#)).

How are we applying a Multi-Head Attention layer in a neural network, where we don't have an arbitrary query, key, and value vector as input? Looking at the computation graph above, a simple but effective implementation is to set

the current feature map in a NN, $X \in \mathbb{R}^{B \times T \times d_{\text{model}}}$, as Q , K and V (B being the batch size, T the sequence length, d_{model} the hidden dimensionality of X). The consecutive weight matrices W^Q , W^K , and W^V can transform X to the corresponding feature vectors that represent the queries, keys, and values of the input. Using this approach, we can implement the Multi-Head Attention module below.

```
[5]: class MultiheadAttention(nn.Module):
    embed_dim : int # Output dimension
    num_heads : int # Number of parallel heads (h)

    def setup(self):
        # Stack all weight matrices 1...h and W^Q, W^K, W^V together for efficiency
        # Note that in many implementations you see "bias=False" which is optional
        self.qkv_proj = nn.Dense(3 * self.embed_dim,
                               kernel_init=nn.initializers.xavier_uniform(), # ←
                               ←Weights with Xavier uniform init
                               bias_init=nn.initializers.zeros # Bias init with zeros
                               )
        self.o_proj = nn.Dense(self.embed_dim,
                               kernel_init=nn.initializers.xavier_uniform(),
                               bias_init=nn.initializers.zeros)

    def __call__(self, x, mask=None):
        batch_size, seq_length, embed_dim = x.shape
        qkv = self.qkv_proj(x)

        # Separate Q, K, V from linear output
        qkv = qkv.reshape(batch_size, seq_length, self.num_heads, -1)
        qkv = qkv.transpose(0, 2, 1, 3) # [Batch, Head, SeqLen, Dims]
        q, k, v = jnp.array_split(qkv, 3, axis=-1)

        # Determine value outputs
        values, attention = scaled_dot_product(q, k, v, mask=mask)
        values = values.transpose(0, 2, 1, 3) # [Batch, SeqLen, Head, Dims]
        values = values.reshape(batch_size, seq_length, embed_dim)
        o = self.o_proj(values)

    return o, attention
```

```
[6]: ## Test MultiheadAttention implementation
# Example features as input
main_rng, x_rng = random.split(main_rng)
x = random.normal(x_rng, (3, 16, 128))
# Create attention
mh_attn = MultiheadAttention(embed_dim=128, num_heads=4)
# Initialize parameters of attention with random key and inputs
main_rng, init_rng = random.split(main_rng)
params = mh_attn.init(init_rng, x)['params']
# Apply attention with parameters on the inputs
out, attn = mh_attn.apply({'params': params}, x)
print('Out', out.shape, 'Attention', attn.shape)

del mh_attn, params
```

Out (3, 16, 128) Attention (3, 4, 16, 16)

One crucial characteristic of the multi-head attention is that it is permutation-equivariant with respect to its inputs. This means that if we switch two input elements in the sequence, e.g. $X_1 \leftrightarrow X_2$ (neglecting the batch dimension for now), the output is exactly the same besides the elements 1 and 2 switched. Hence, the multi-head attention is actually looking at the input not as a sequence, but as a set of elements. This property makes the multi-head attention block and the Transformer architecture so powerful and widely applicable! But what if the order of the input is actually important for solving the task, like language modeling? The answer is to encode the position in the input features, which we will take a closer look at later (topic *Positional encodings* below).

Before moving on to creating the Transformer architecture, we can compare the self-attention operation with our other common layer competitors for sequence data: convolutions and recurrent neural networks. Below you can find a table by [Vaswani et al. \(2017\)](#) on the complexity per layer, the number of sequential operations, and maximum path length. The complexity is measured by the upper bound of the number of operations to perform, while the maximum path length represents the maximum number of steps a forward or backward signal has to traverse to reach any other position. The lower this length, the better gradient signals can backpropagate for long-range dependencies. Let's take a look at the table below:

n is the sequence length, d is the representation dimension and k is the kernel size of convolutions. In contrast to recurrent networks, the self-attention layer can parallelize all its operations making it much faster to execute for smaller sequence lengths. However, when the sequence length exceeds the hidden dimensionality, self-attention becomes more expensive than RNNs. One way of reducing the computational cost for long sequences is by restricting the self-attention to a neighborhood of inputs to attend over, denoted by r . Nevertheless, there has been recently a lot of work on more efficient Transformer architectures that still allow long dependencies, of which you can find an overview in the paper by [Tay et al. \(2020\)](#) if interested.

Transformer Encoder

Next, we will look at how to apply the multi-head attention block inside the Transformer architecture. Originally, the Transformer model was designed for machine translation. Hence, it got an encoder-decoder structure where the encoder takes as input the sentence in the original language and generates an attention-based representation. On the other hand, the decoder attends over the encoded information and generates the translated sentence in an autoregressive manner, as in a standard RNN. While this structure is extremely useful for Sequence-to-Sequence tasks with the necessity of autoregressive decoding, we will focus here on the encoder part. Many advances in NLP have been made using pure encoder-based Transformer models (if interested, models include the [BERT](#)-family, the [Vision Transformer](#), and more), and in our tutorial, we will also mainly focus on the encoder part. If you have understood the encoder architecture, the decoder is a very small step to implement as well. The full Transformer architecture looks as follows (figure credit - [Vaswani et al., 2017](#)).:

The encoder consists of N identical blocks that are applied in sequence. Taking as input x , it is first passed through a Multi-Head Attention block as we have implemented above. The output is added to the original input using a residual connection, and we apply a consecutive Layer Normalization on the sum. Overall, it calculates $\text{LayerNorm}(x + \text{Multihead}(x, x, x))$ (x being Q , K and V input to the attention layer). The residual connection is crucial in the Transformer architecture for two reasons:

1. Similar to ResNets, Transformers are designed to be very deep. Some models contain more than 24 blocks in the encoder. Hence, the residual connections are crucial for enabling a smooth gradient flow through the model.
2. Without the residual connection, the information about the original sequence is lost. Remember that the Multi-Head Attention layer ignores the position of elements in a sequence, and can only learn it based on the input features. Removing the residual connections would mean that this information is lost after the first attention layer (after initialization), and with a randomly initialized query and key vector, the output vectors for position i

has no relation to its original input. All outputs of the attention are likely to represent similar/same information, and there is no chance for the model to distinguish which information came from which input element. An alternative option to residual connection would be to fix at least one head to focus on its original input, but this is very inefficient and does not have the benefit of the improved gradient flow.

The Layer Normalization also plays an important role in the Transformer architecture as it enables faster training and provides small regularization. Additionally, it ensures that the features are in a similar magnitude among the elements in the sequence. We are not using Batch Normalization because it depends on the batch size which is often small with Transformers (they require a lot of GPU memory), and BatchNorm has shown to perform particularly bad in language as the features of words tend to have a much higher variance (there are many, very rare words which need to be considered for a good distribution estimate).

Additionally to the Multi-Head Attention, a small fully connected feed-forward network is added to the model, which is applied to each position separately and identically. Specifically, the model uses a Linear→ReLU→Linear MLP. The full transformation including the residual connection can be expressed as:

$$\begin{aligned} \text{FFN}(x) &= \max(0, xW_1 + b_1)W_2 + b_2 \\ x &= \text{LayerNorm}(x + \text{FFN}(x)) \end{aligned}$$

This MLP adds extra complexity to the model and allows transformations on each sequence element separately. You can imagine as this allows the model to “post-process” the new information added by the previous Multi-Head Attention, and prepare it for the next attention block. Usually, the inner dimensionality of the MLP is $2\text{-}8\times$ larger than d_{model} , i.e. the dimensionality of the original input x . The general advantage of a wider layer instead of a narrow, multi-layer MLP is the faster, parallelizable execution.

Finally, after looking at all parts of the encoder architecture, we can start implementing it below. We first start by implementing a single encoder block. Additionally to the layers described above, we will add dropout layers in the MLP and on the output of the MLP and Multi-Head Attention for regularization.

```
[7]: class EncoderBlock(nn.Module):
    input_dim : int # Input dimension is needed here since it is equal to the output_
    ↪dimension (residual connection)
    num_heads : int
    dim_feedforward : int
    dropout_prob : float

    def setup(self):
        # Attention layer
        self.self_attn = MultiheadAttention(embed_dim=self.input_dim,
                                             num_heads=self.num_heads)
        # Two-layer MLP
        self.linear = [
            nn.Dense(self.dim_feedforward),
            nn.Dropout(self.dropout_prob),
            nn.relu,
            nn.Dense(self.input_dim)
        ]
        # Layers to apply in between the main layers
        self.norm1 = nn.LayerNorm()
        self.norm2 = nn.LayerNorm()
        self.dropout = nn.Dropout(self.dropout_prob)

    def __call__(self, x, mask=None, train=True):
        # Attention part
        attn_out, _ = self.self_attn(x, mask=mask)
```

(continues on next page)

(continued from previous page)

```

x = x + self.dropout(attn_out, deterministic=not train)
x = self.norm1(x)

# MLP part
linear_out = x
for l in self.linear:
    linear_out = l(linear_out) if not isinstance(l, nn.Dropout) else l(linear_
out, deterministic=not train)
    x = x + self.dropout(linear_out, deterministic=not train)
x = self.norm2(x)

return x

```

[8]:

```

## Test EncoderBlock implementation
# Example features as input
main_rng, x_rng = random.split(main_rng)
x = random.normal(x_rng, (3, 16, 128))
# Create encoder block
encblock = EncoderBlock(input_dim=128, num_heads=4, dim_feedforward=512, dropout_prob=0.
↪1)
# Initialize parameters of encoder block with random key and inputs
main_rng, init_rng, dropout_init_rng = random.split(main_rng, 3)
params = encblock.init({'params': init_rng, 'dropout': dropout_init_rng}, x, True)[
↪'params']
# Apply encoder block with parameters on the inputs
# Since dropout is stochastic, we need to pass a rng to the forward
main_rng, dropout_apply_rng = random.split(main_rng)
out = encblock.apply({'params': params}, x, train=True, rngs={'dropout': dropout_apply_↪rng})
print('Out', out.shape)

del encblock, params

```

Out (3, 16, 128)

Based on this block, we can implement a module for the full Transformer encoder. Additionally to a forward function that iterates through the sequence of encoder blocks, we also provide a function called `get_attention_maps`. The idea of this function is to return the attention probabilities for all Multi-Head Attention blocks in the encoder. This helps us in understanding, and in a sense, explaining the model. However, the attention probabilities should be interpreted with a grain of salt as it does not necessarily reflect the true interpretation of the model (there is a series of papers about this, including [Attention is not Explanation](#) and [Attention is not not Explanation](#)).

[9]:

```

class TransformerEncoder(nn.Module):
    num_layers : int
    input_dim : int
    num_heads : int
    dim_feedforward : int
    dropout_prob : float

    def setup(self):
        self.layers = [EncoderBlock(self.input_dim, self.num_heads, self.dim_feedforward,
↪ self.dropout_prob) for _ in range(self.num_layers)]

```

(continues on next page)

(continued from previous page)

```

def __call__(self, x, mask=None, train=True):
    for l in self.layers:
        x = l(x, mask=mask, train=train)
    return x

def get_attention_maps(self, x, mask=None, train=True):
    # A function to return the attention maps within the model for a single
    # application
    # Used for visualization purpose later
    attention_maps = []
    for l in self.layers:
        _, attn_map = l.self_attn(x, mask=mask)
        attention_maps.append(attn_map)
        x = l(x, mask=mask, train=train)
    return attention_maps

```

```

[10]: ## Test TransformerEncoder implementation
# Example features as input
main_rng, x_rng = random.split(main_rng)
x = random.normal(x_rng, (3, 16, 128))
# Create Transformer encoder
transenc = TransformerEncoder(num_layers=5,
                               input_dim=128,
                               num_heads=4,
                               dim_feedforward=256,
                               dropout_prob=0.15)
# Initialize parameters of transformer with random key and inputs
main_rng, init_rng, dropout_init_rng = random.split(main_rng, 3)
params = transenc.init({'params': init_rng, 'dropout': dropout_init_rng}, x, train=True)[
    'params']
# Apply transformer with parameters on the inputs
# Since dropout is stochastic, we need to pass a rng to the forward
main_rng, dropout_apply_rng = random.split(main_rng)
# Instead of passing params and rngs every time to a function call, we can bind them to
# the module
binded_mod = transenc.bind({'params': params}, rngs={'dropout': dropout_apply_rng})
out = binded_mod(x, train=True)
print('Out', out.shape)
attn_maps = binded_mod.get_attention_maps(x, train=True)
print('Attention maps', len(attn_maps), attn_maps[0].shape)

del transenc, binded_mod, params

```

Out (3, 16, 128)
Attention maps 5 (3, 4, 16, 16)

Positional encoding

We have discussed before that the Multi-Head Attention block is permutation-equivariant, and cannot distinguish whether an input comes before another one in the sequence or not. In tasks like language understanding, however, the position is important for interpreting the input words. The position information can therefore be added via the input features. We could learn a embedding for every possible position, but this would not generalize to a dynamical input sequence length. Hence, the better option is to use feature patterns that the network can identify from the features and potentially generalize to larger sequences. The specific pattern chosen by Vaswani et al. are sine and cosine functions of different frequencies, as follows:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{\text{model}}}}\right) & \text{if } i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{\text{model}}}}\right) & \text{otherwise} \end{cases}$$

$PE_{(pos,i)}$ represents the position encoding at position pos in the sequence, and hidden dimensionality i . These values, concatenated for all hidden dimensions, are added to the original input features (in the Transformer visualization above, see “Positional encoding”), and constitute the position information. We distinguish between even ($i \bmod 2 = 0$) and uneven ($i \bmod 2 = 1$) hidden dimensionalities where we apply a sine/cosine respectively. The intuition behind this encoding is that you can represent $PE_{(pos+k,:)}$ as a linear function of $PE_{(pos,:)}$, which might allow the model to easily attend to relative positions. The wavelengths in different dimensions range from 2π to $10000 \cdot 2\pi$.

The positional encoding is implemented below.

```
[11]: class PositionalEncoding(nn.Module):
    d_model : int          # Hidden dimensionality of the input.
    max_len : int = 5000   # Maximum length of a sequence to expect.

    def setup(self):
        # Create matrix of [SeqLen, HiddenDim] representing the positional encoding for
        ↵max_len inputs
        pe = np.zeros((self.max_len, self.d_model))
        position = np.arange(0, self.max_len, dtype=np.float32)[:,None]
        div_term = np.exp(np.arange(0, self.d_model, 2) * (-math.log(10000.0) / self.d_
        ↵model)))
        pe[:, 0::2] = np.sin(position * div_term)
        pe[:, 1::2] = np.cos(position * div_term)
        pe = pe[None]
        self.pe = jax.device_put(pe)

    def __call__(self, x):
        x = x + self.pe[:, :x.shape[1]]
        return x
```

To understand the positional encoding, we can visualize it below. We will generate an image of the positional encoding over hidden dimensionality and position in a sequence. Each pixel, therefore, represents the change of the input feature we perform to encode the specific position. Let's do it below.

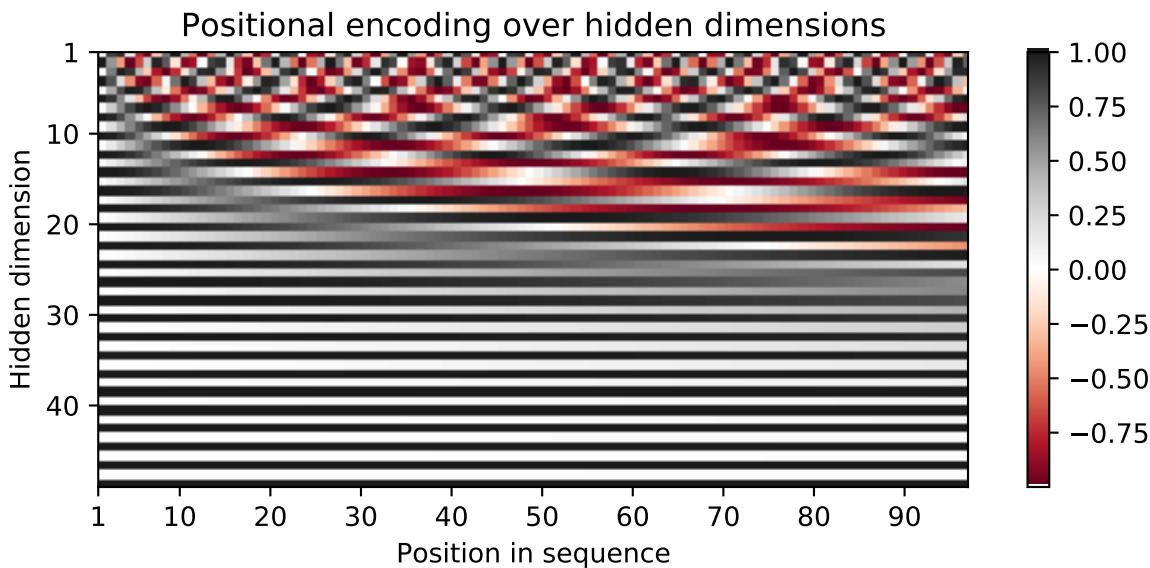
```
[12]: # Create encoding block, bind to access positional encoding (module has no parameters)
encod_block = PositionalEncoding(d_model=48, max_len=96).bind({})
# Obtain positional encodings as numpy array
pe = jax.device_get(encod_block.pe.squeeze().T)

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,3))
pos = ax.imshow(pe, cmap="RdGy", extent=(1,pe.shape[1]+1,pe.shape[0]+1,1))
fig.colorbar(pos, ax=ax)
```

(continues on next page)

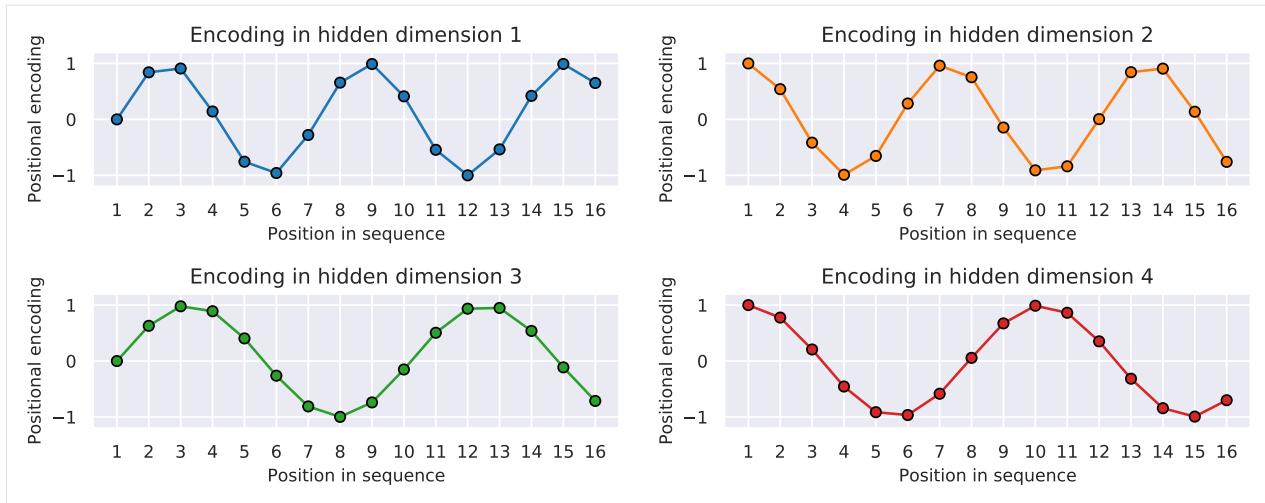
(continued from previous page)

```
ax.set_xlabel("Position in sequence")
ax.set_ylabel("Hidden dimension")
ax.set_title("Positional encoding over hidden dimensions")
ax.set_xticks([1]+[i*10 for i in range(1,1+pe.shape[1]//10)])
ax.set_yticks([1]+[i*10 for i in range(1,1+pe.shape[0]//10)])
plt.show()
```



You can clearly see the sine and cosine waves with different wavelengths that encode the position in the hidden dimensions. Specifically, we can look at the sine/cosine wave for each hidden dimension separately, to get a better intuition of the pattern. Below we visualize the positional encoding for the hidden dimensions 1, 2, 3 and 4.

```
[13]: sns.set_theme()
fig, ax = plt.subplots(2, 2, figsize=(12, 4))
ax = [a for a_list in ax for a in a_list]
for i in range(len(ax)):
    ax[i].plot(np.arange(1,17), pe[i,:16], color=f'C{i}', marker="o", markersize=6, markeredgewidth=1)
    ax[i].set_title(f"Encoding in hidden dimension {i+1}")
    ax[i].set_xlabel("Position in sequence", fontsize=10)
    ax[i].set_ylabel("Positional encoding", fontsize=10)
    ax[i].set_xticks(np.arange(1,17))
    ax[i].tick_params(axis='both', which='major', labelsize=10)
    ax[i].tick_params(axis='both', which='minor', labelsize=8)
    ax[i].set_ylim(-1.2, 1.2)
fig.subplots_adjust(hspace=0.8)
sns.reset_orig()
plt.show()
```



As we can see, the patterns between the hidden dimension 1 and 2 only differ in the starting angle. The wavelength is 2π , hence the repetition after position 6. The hidden dimensions 2 and 3 have about twice the wavelength.

Learning rate warm-up

One commonly used technique for training a Transformer is learning rate warm-up. This means that we gradually increase the learning rate from 0 on to our originally specified learning rate in the first few iterations. Thus, we slowly start learning instead of taking very large steps from the beginning. In fact, training a deep Transformer without learning rate warm-up can make the model diverge and achieve a much worse performance on training and testing. Take for instance the following plot by Liu et al. (2019) comparing Adam-vanilla (i.e. Adam without warm-up) vs Adam with a warm-up:

Clearly, the warm-up is a crucial hyperparameter in the Transformer architecture. Why is it so important? There are currently two common explanations. Firstly, Adam uses the bias correction factors which however can lead to a higher variance in the adaptive learning rate during the first iterations. Improved optimizers like RAdam have been shown to overcome this issue, not requiring warm-up for training Transformers. Secondly, the iteratively applied Layer Normalization across layers can lead to very high gradients during the first iterations, which can be solved by using Pre-Layer Normalization (similar to Pre-Activation ResNet), or replacing Layer Normalization by other techniques (Adaptive Normalization, Power Normalization).

Nevertheless, many applications and papers still use the original Transformer architecture with Adam, because warm-up is a simple, yet effective way of solving the gradient problem in the first iterations. There are many different schedulers we could use. For instance, the original Transformer paper used an exponential decay scheduler with a warm-up. However, the currently most popular scheduler is the cosine warm-up scheduler, which combines warm-up with a cosine-shaped learning rate decay. In Optax, this learning rate scheduler is also implemented in `optax.warmup_cosine_decay_schedule`, but let's manually implement it below, and visualize the learning rate factor over epochs.

```
[14]: def cosine_warmup_schedule(base_lr: float, warmup: int, max_iters: int):
    assert warmup > 0 and max_iters > 0
    # Create function to return lr based on iteration count
    def get_lr(train_iter):
        lr_factor = 0.5 * (1 + np.cos(np.pi * train_iter / max_iters))
        if train_iter <= warmup:
            lr_factor *= train_iter * 1.0 / warmup
    return get_lr
```

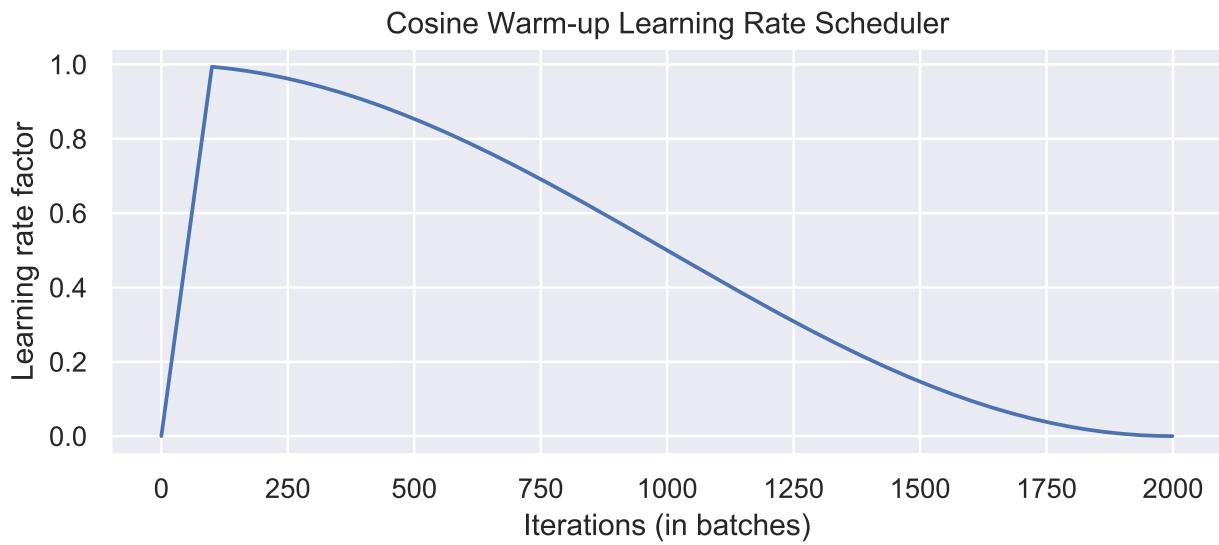
(continues on next page)

(continued from previous page)

```
    return lr_factor * base_lr
return get_lr
```

[15]: lr_scheduler = cosine_warmup_schedule(base_lr=1.0, warmup=100, max_iters=2000)

```
# Plotting
epochs = list(range(2000))
sns.set()
plt.figure(figsize=(8,3))
plt.plot(epochs, [lr_scheduler(e) for e in epochs])
plt.ylabel("Learning rate factor")
plt.xlabel("Iterations (in batches)")
plt.title("Cosine Warm-up Learning Rate Scheduler")
plt.show()
sns.reset_orig()
```



In the first 100 iterations, we increase the learning rate factor from 0 to 1, whereas for all later iterations, we decay it using the cosine wave.

Full Transformer model

Finally, we can embed the Transformer architecture into a full architecture. We will implement a template for a classifier based on the Transformer encoder. Thereby, we have a prediction output per sequence element. If we would need a classifier over the whole sequence, the common approach is to add an additional [CLS] token to the sequence, representing the classifier token. However, here we focus on tasks where we have one output per element.

Additionally to the Transformer architecture, we add a small input network (maps input dimensions to model dimensions), the positional encoding, and an output network (transforms output encodings to predictions). We also add the learning rate scheduler, which takes a step each iteration instead of once per epoch. This is needed for the warmup and the smooth cosine decay. The training, validation, and test step is left empty for now and will be filled for our task-specific models.

```
[16]: class TransformerPredictor(nn.Module):
    model_dim : int # Hidden dimensionality to use inside the Transformer
    ↵Transformer
    num_classes : int # Number of classes to predict per sequence element
    num_heads : int # Number of heads to use in the Multi-Head
    ↵Attention blocks
    num_layers : int # Number of encoder blocks to use
    dropout_prob : float = 0.0 # Dropout to apply inside the model
    input_dropout_prob : float = 0.0 # Dropout to apply on the input features

    def setup(self):
        # Input dim -> Model dim
        self.input_dropout = nn.Dropout(self.input_dropout_prob)
        self.input_layer = nn.Dense(self.model_dim)
        # Positional encoding for sequences
        self.positional_encoding = PositionalEncoding(self.model_dim)
        # Transformer
        self.transformer = TransformerEncoder(num_layers=self.num_layers,
                                              input_dim=self.model_dim,
                                              dim_feedforward=2*self.model_dim,
                                              num_heads=self.num_heads,
                                              dropout_prob=self.dropout_prob)

        # Output classifier per sequence element
        self.output_net = [
            nn.Dense(self.model_dim),
            nn.LayerNorm(),
            nn.relu,
            nn.Dropout(self.dropout_prob),
            nn.Dense(self.num_classes)
        ]

    def __call__(self, x, mask=None, add_positional_encoding=True, train=True):
        """
        Inputs:
            x - Input features of shape [Batch, SeqLen, input_dim]
            mask - Mask to apply on the attention outputs (optional)
            add_positional_encoding - If True, we add the positional encoding to the input.
        """
        x = self.input_dropout(x, deterministic=not train)
        x = self.input_layer(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        x = self.transformer(x, mask=mask, train=train)
        for l in self.output_net:
            x = l(x) if not isinstance(l, nn.Dropout) else l(x, deterministic=not train)
        return x

    def get_attention_maps(self, x, mask=None, add_positional_encoding=True, train=True):
        """
        Function for extracting the attention matrices of the whole Transformer for a single batch.
        """

```

(continues on next page)

(continued from previous page)

```
Input arguments same as the forward pass.
"""
x = self.input_dropout(x, deterministic=not train)
x = self.input_layer(x)
if add_positional_encoding:
    x = self.positional_encoding(x)
attention_maps = self.transformer.get_attention_maps(x, mask=mask, train=train)
return attention_maps
```

```
[17]: ## Test TransformerPredictor implementation
# Example features as input
main_rng, x_rng = random.split(main_rng)
x = random.normal(x_rng, (3, 16, 64))
# Create Transformer encoder
transpre = TransformerPredictor(num_layers=5,
                                model_dim=128,
                                num_classes=10,
                                num_heads=4,
                                dropout_prob=0.15,
                                input_dropout_prob=0.05)
# Initialize parameters of transformer predictor with random key and inputs
main_rng, init_rng, dropout_init_rng = random.split(main_rng, 3)
params = transpre.init({'params': init_rng, 'dropout': dropout_init_rng}, x, train=True)[
    'params']
# Apply transformer predictor with parameters on the inputs
# Since dropout is stochastic, we need to pass a rng to the forward
main_rng, dropout_apply_rng = random.split(main_rng)
# Instead of passing params and rngs every time to a function call, we can bind them to
# the module
binded_mod = transpre.bind({'params': params}, rngs={'dropout': dropout_apply_rng})
out = binded_mod(x, train=True)
print('Out', out.shape)
attn_maps = binded_mod.get_attention_maps(x, train=True)
print('Attention maps', len(attn_maps), attn_maps[0].shape)

del transpre, binded_mod, params
Out (3, 16, 10)
Attention maps 5 (3, 4, 16, 16)
```

Trainer module

Finally, we add the missing parts needed for training a model in JAX and Flax. Note that we leave the specific loss function unimplemented, since this function depends on different tasks we do below.

In the optimization, we use the Adam optimizer with our previously discussed cosine scheduler. Additionally, we use the optax transformation `optax.clip_by_global_norm` ([documentation](#)). This clips the norm of the gradients for all parameters before taking an optimizer step and prevents the model from diverging if we obtain very high gradients at, for instance, sharp loss surfaces (see many good blog posts on gradient clipping, like [DeepAI glossary](#)). For Transformers, gradient clipping can help to further stabilize the training during the first few iterations, and also afterward. The clip value is usually between 0.5 and 10, depending on how harsh you want to clip large gradients.

[18]: `class TrainerModule:`

```

    def __init__(self, model_name, exmp_batch, max_iters, lr=1e-3, warmup=100, seed=42,_
     ↪ **model_kwargs):
        """
        Inputs:
            model_name - Name of the model. Used for saving and checkpointing
            exmp_batch - Example batch to the model for initialization
            max_iters - Number of maximum iterations the model is trained for. This is_
             ↪ needed for the CosineWarmup scheduler
            lr - Learning rate in the optimizer
            warmup - Number of warmup steps. Usually between 50 and 500
            seed - Seed to use for model init
        """
        super().__init__()
        self.model_name = model_name
        self.max_iters = max_iters
        self.lr = lr
        self.warmup = warmup
        self.seed = seed
        # Create empty model. Note: no parameters yet
        self.model = TransformerPredictor(**model_kwargs)
        # Prepare logging
        self.log_dir = os.path.join(CHECKPOINT_PATH, self.model_name)
        self.logger = SummaryWriter(log_dir=self.log_dir)
        # Create jitted training and eval functions
        self.create_functions()
        # Initialize model
        self.init_model(exmp_batch)

    def batch_to_input(self, exmp_batch):
        # Map batch to input data to the model
        # To be implemented in a task specific sub-class
        raise NotImplementedError

    def get_loss_function(self):
        # Return a function that calculates the loss for a batch
        # To be implemented in a task specific sub-class
        raise NotImplementedError

    def create_functions(self):
        # Create jitted train and eval functions
        calculate_loss = self.get_loss_function()

        # Training function
        def train_step(state, rng, batch):
            loss_fn = lambda params: calculate_loss(params, rng, batch, train=True)
            ret, grads = jax.value_and_grad(loss_fn, has_aux=True)(state.params)
            loss, acc, rng = ret[0], *ret[1]
            state = state.apply_gradients(grads=grads)
            return state, rng, loss, acc
        self.train_step = jax.jit(train_step)

```

(continues on next page)

(continued from previous page)

```

# Evaluation function
def eval_step(state, rng, batch):
    _, (acc, rng) = calculate_loss(state.params, rng, batch, train=False)
    return acc, rng
self.eval_step = jax.jit(eval_step)

def init_model(self, exmp_batch):
    # Initialize model
    self.rng = jax.random.PRNGKey(self.seed)
    self.rng, init_rng, dropout_init_rng = jax.random.split(self.rng, 3)
    exmp_input = self.batch_to_input(exmp_batch)
    params = self.model.init({'params': init_rng, 'dropout': dropout_init_rng}, exmp_
    ↵input, train=True)['params']
    # Initialize learning rate schedule and optimizer
    lr_schedule = optax.warmup_cosine_decay_schedule(
        init_value=0.0,
        peak_value=self.lr,
        warmup_steps=self.warmup,
        decay_steps=self.max_iters,
        end_value=0.0
    )
    optimizer = optax.chain(
        optax.clip_by_global_norm(1.0), # Clip gradients at norm 1
        optax.adam(lr_schedule)
    )
    # Initialize training state
    self.state = train_state.TrainState.create(apply_fn=self.model.apply,_
    ↵params=params, tx=optimizer)

def train_model(self, train_loader, val_loader, num_epochs=500):
    # Train model for defined number of epochs
    best_acc = 0.0
    for epoch_idx in tqdm(range(1, num_epochs+1)):
        self.train_epoch(train_loader, epoch=epoch_idx)
        if epoch_idx % 5 == 0:
            eval_acc = self.eval_model(val_loader)
            self.logger.add_scalar('val/accuracy', eval_acc, global_step=epoch_idx)
            if eval_acc >= best_acc:
                best_acc = eval_acc
                self.save_model(step=epoch_idx)
            self.logger.flush()

def train_epoch(self, train_loader, epoch):
    # Train model for one epoch, and log avg loss and accuracy
    accs, losses = [], []
    for batch in tqdm(train_loader, desc='Training', leave=False):
        self.state, self.rng, loss, accuracy = self.train_step(self.state, self.rng,_
        ↵batch)
        losses.append(loss)
        accs.append(accuracy)
    avg_loss = np.stack(jax.device_get(losses)).mean()
    avg_acc = np.stack(jax.device_get(accs)).mean()

```

(continues on next page)

(continued from previous page)

```

self.logger.add_scalar('train/loss', avg_loss, global_step=epoch)
self.logger.add_scalar('train/accuracy', avg_acc, global_step=epoch)

def eval_model(self, data_loader):
    # Test model on all data points of a data loader and return avg accuracy
    correct_class, count = 0, 0
    for batch in data_loader:
        acc, self.rng = self.eval_step(self.state, self.rng, batch)
        correct_class += acc * batch[0].shape[0]
        count += batch[0].shape[0]
    eval_acc = (correct_class / count).item()
    return eval_acc

def save_model(self, step=0):
    # Save current model at certain training iteration
    checkpoints.save_checkpoint(ckpt_dir=self.log_dir, target=self.state.params,
                                step=step)

def load_model(self, pretrained=False):
    # Load model. We use different checkpoint for the pretrained model
    if not pretrained:
        params = checkpoints.restore_checkpoint(ckpt_dir=self.log_dir, target=self.state.params)
    else:
        params = checkpoints.restore_checkpoint(ckpt_dir=os.path.join(CHECKPOINT_PATH,
                                                                     f'{self.model_name}.ckpt'), target=self.state.params)
        self.state = TrainState.create(apply_fn=self.model.apply, params=params, tx=self.state.tx)

def checkpoint_exists(self):
    # Check whether a pretrained model exist for this Transformer
    return os.path.isfile(os.path.join(CHECKPOINT_PATH, f'{self.model_name}.ckpt'))

```

4.23.2 Experiments

After having finished the implementation of the Transformer architecture, we can start experimenting and apply it to various tasks. In this notebook, we will focus on two tasks: parallel Sequence-to-Sequence, and set anomaly detection. The two tasks focus on different properties of the Transformer architecture, and we go through them below.

Sequence to Sequence

A Sequence-to-Sequence task represents a task where the input *and* the output is a sequence, not necessarily of the same length. Popular tasks in this domain include machine translation and summarization. For this, we usually have a Transformer encoder for interpreting the input sequence, and a decoder for generating the output in an autoregressive manner. Here, however, we will go back to a much simpler example task and use only the encoder. Given a sequence of N numbers between 0 and M , the task is to reverse the input sequence. In Numpy notation, if our input is x , the output should be $x[::-1]$. Although this task sounds very simple, RNNs can have issues with such because the task requires long-term dependencies. Transformers are built to support such, and hence, we expect it to perform very well.

First, let's create a dataset class below.

```
[19]: class ReverseDataset(data.Dataset):

    def __init__(self, num_categories, seq_len, size, np_rng):
        super().__init__()
        self.num_categories = num_categories
        self.seq_len = seq_len
        self.size = size
        self.np_rng = np_rng

        self.data = self.np_rng.integers(self.num_categories, size=(self.size, self.seq_
len))

    def __len__(self):
        return self.size

    def __getitem__(self, idx):
        inp_data = self.data[idx]
        labels = np.flip(inp_data, axis=0)
        return inp_data, labels
```

We create an arbitrary number of random sequences of numbers between 0 and `num_categories`-1. The label is simply the tensor flipped over the sequence dimension. We can create the corresponding data loaders using PyTorch below.

```
[20]: # Combine batch elements (all numpy) by stacking
def numpy_collate(batch):
    if isinstance(batch[0], np.ndarray):
        return np.stack(batch)
    elif isinstance(batch[0], (tuple, list)):
        transposed = zip(*batch)
        return [numpy_collate(samples) for samples in transposed]
    else:
        return np.array(batch)

dataset = partial(ReverseDataset, 10, 16)
rev_train_loader = data.DataLoader(dataset(50000, np_rng=np.random.default_rng(42)),
                                    batch_size=128,
                                    shuffle=True,
                                    drop_last=True,
                                    collate_fn=numpy_collate)
rev_val_loader = data.DataLoader(dataset(1000, np_rng=np.random.default_rng(43)),
                                 batch_size=128,
                                 collate_fn=numpy_collate)
rev_test_loader = data.DataLoader(dataset(10000, np_rng=np.random.default_rng(44)),
                                  batch_size=128,
                                  collate_fn=numpy_collate)
```

Remember that these data loaders return numpy arrays instead of PyTorch tensors, as we define in the `numpy_collate` function which combines individual elements to a batch. As the data set is so simple and the `__getitem__` finishes a neglectable time, we don't need subprocesses, i.e. workers, to provide us the data (in fact, more workers can slow down the training as we have communication overhead among processes/threads).

Now, let's look at an arbitrary sample of the dataset:

```
[21]: inp_data, labels = rev_train_loader.dataset[0]
print("Input data:", inp_data)
print("Labels:    ", labels)

Input data: [0 7 6 4 4 8 0 6 2 0 5 9 7 7 7 7]
Labels:      [7 7 7 7 9 5 0 2 6 0 8 4 4 6 7 0]
```

During training, we pass the input sequence through the Transformer encoder and predict the output for each input token. We use the standard Cross-Entropy loss to perform this. Every number is represented as a one-hot vector. Remember that representing the categories as single scalars decreases the expressiveness of the model extremely as 0 and 1 are not closer related than 0 and 9 in our example. An alternative to a one-hot vector is using a learned embedding vector as it is provided by an `nn.Embed` module ([documentation](#)). However, using a one-hot vector with an additional linear layer as in our case has the same effect as an embedding layer (`self.input_net` maps one-hot vector to a dense vector, where each row of the weight matrix represents the embedding for a specific category).

To implement the training dynamic, we create a new class inheriting from `TrainerModule` and defining the loss function as follows:

```
[22]: class ReverseTrainer(TrainerModule):

    def batch_to_input(self, batch):
        inp_data, _ = batch
        inp_data = jax.nn.one_hot(inp_data, num_classes=self.model.num_classes)
        return inp_data

    def get_loss_function(self):
        # Function for calculating loss and accuracy for a batch
        def calculate_loss(params, rng, batch, train):
            inp_data, labels = batch
            inp_data = jax.nn.one_hot(inp_data, num_classes=self.model.num_classes)
            rng, dropout_apply_rng = random.split(rng)
            logits = self.model.apply({'params': params}, inp_data, train=train, rngs={
                'dropout': dropout_apply_rng})
            loss = optax.softmax_cross_entropy_with_integer_labels(logits, labels).mean()
            acc = (logits.argmax(axis=-1) == labels).mean()
            return loss, (acc, rng)
        return calculate_loss
```

Finally, we can create a training function, similar to ones we have seen before. We create a `ReverseTrainer` object, run the training for N epochs while logging in TensorBoard, and saving our best model based on the validation. Afterward, we test our models on the test set.

```
[23]: def train_reverse(max_epochs=10, **model_args):
    num_train_iters = len(rev_train_loader) * max_epochs
    # Create a trainer module with specified hyperparameters
    trainer = ReverseTrainer(model_name='ReverseTask',
                             exmp_batch=next(iter(rev_train_loader)),
                             max_iters=num_train_iters,
                             **model_args)
    if not trainer.checkpoint_exists(): # Skip training if pretrained model exists
        trainer.train_model(rev_train_loader, rev_val_loader, num_epochs=max_epochs)
        trainer.load_model()
    else:
        trainer.load_model(pretrained=True)
```

(continues on next page)

(continued from previous page)

```

val_acc = trainer.eval_model(rev_val_loader)
test_acc = trainer.eval_model(rev_test_loader)
# Bind parameters to model for easier inference
trainer.model_bd = trainer.model.bind({'params': trainer.state.params})
return trainer, {'val_acc': val_acc, 'test_acc': test_acc}

```

Finally, we can train the model. In this setup, we will use a single encoder block and a single head in the Multi-Head Attention. This is chosen because of the simplicity of the task, and in this case, the attention can actually be interpreted as an “explanation” of the predictions (compared to the other papers above dealing with deep Transformers).

```
[24]: reverse_trainer, reverse_result = train_reverse(model_dim=32,
                                                    num_heads=1,
                                                    num_classes=rev_train_loader.dataset.num_
←categories,
                                                    num_layers=1,
                                                    dropout_prob=0.0,
                                                    lr=5e-4,
                                                    warmup=50)
```

First, let's print the results:

```
[25]: print(f"Val accuracy: {(100.0 * reverse_result['val_acc']):.2f}%")
print(f"Test accuracy: {(100.0 * reverse_result['test_acc']):.2f}%")

Val accuracy: 100.00%
Test accuracy: 100.00%
```

As we would have expected, the Transformer can correctly solve the task. However, how does the attention in the Multi-Head Attention block looks like for an arbitrary input? Let's try to visualize it below.

```
[26]: data_input, labels = next(iter(rev_val_loader))
inp_data = jax.nn.one_hot(data_input, num_classes=reverse_trainer.model.num_classes)
attention_maps = reverse_trainer.model_bd.get_attention_maps(inp_data)
```

The object `attention_maps` is a list of length N where N is the number of layers. Each element is a tensor of shape [Batch, Heads, SeqLen, SeqLen], which we can verify below.

```
[27]: attention_maps[0].shape
[27]: (128, 1, 16, 16)
```

Next, we will write a plotting function that takes as input the sequences, attention maps, and an index indicating for which batch element we want to visualize the attention map. We will create a plot where over rows, we have different layers, while over columns, we show the different heads. Remember that the softmax has been applied for each row separately.

```
[28]: def plot_attention_maps(input_data, attn_maps, idx=0):
    if input_data is not None:
        input_data = jax.device_get(input_data[idx])
    else:
        input_data = np.arange(attn_maps[0][idx].shape[-1])
    attn_maps = [jax.device_get(m[idx]) for m in attn_maps]

    num_heads = attn_maps[0].shape[0]
```

(continues on next page)

(continued from previous page)

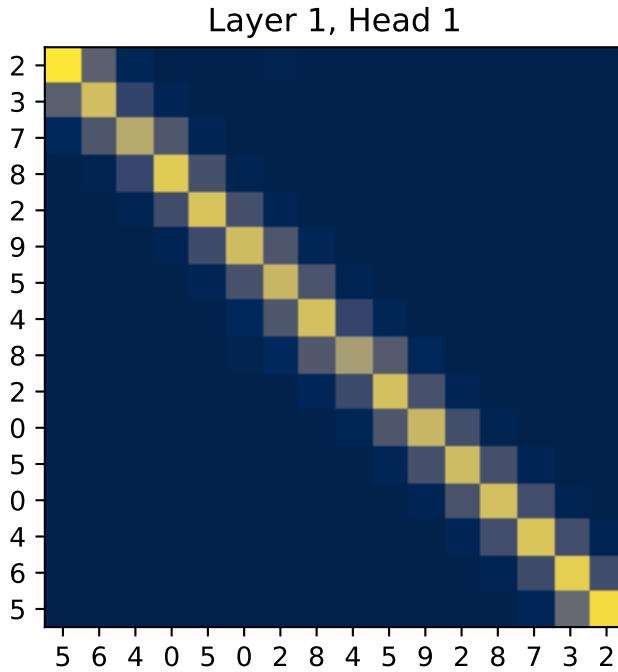
```

num_layers = len(attn_maps)
seq_len = input_data.shape[0]
fig_size = 4 if num_heads == 1 else 3
fig, ax = plt.subplots(num_layers, num_heads, figsize=(num_heads*fig_size, num_
layers*fig_size))
if num_layers == 1:
    ax = [ax]
if num_heads == 1:
    ax = [[a] for a in ax]
for row in range(num_layers):
    for column in range(num_heads):
        ax[row][column].imshow(attn_maps[row][column], origin='lower', vmin=0)
        ax[row][column].set_xticks(list(range(seq_len)))
        ax[row][column].set_xticklabels(input_data.tolist())
        ax[row][column].set_yticks(list(range(seq_len)))
        ax[row][column].set_yticklabels(input_data.tolist())
        ax[row][column].set_title(f"Layer {row+1}, Head {column+1}")
fig.subplots_adjust(hspace=0.5)
plt.show()

```

Finally, we can plot the attention map of our trained Transformer on the reverse task:

[29]: `plot_attention_maps(data_input, attention_maps, idx=0)`

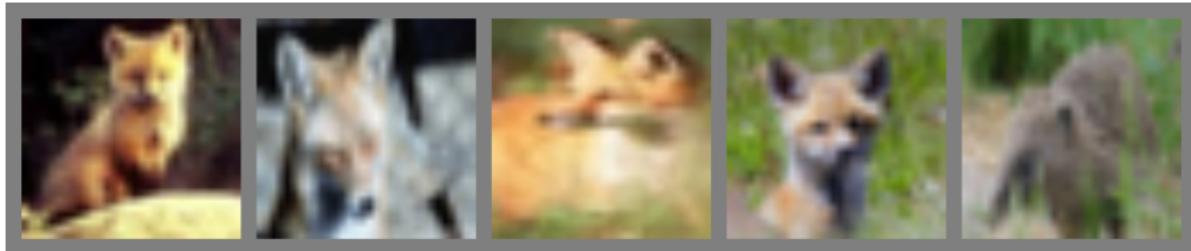


The model has learned to attend to the token that is on the flipped index of itself. Hence, it actually does what we intended it to do. We see that it however also pays some attention to values close to the flipped index. This is because the model doesn't need the perfect, hard attention to solve this problem, but is fine with this approximate, noisy attention map. The close-by indices are caused by the similarity of the positional encoding, which we also intended with the positional encoding.

Set Anomaly Detection

Besides sequences, sets are another data structure that is relevant for many applications. In contrast to sequences, elements are unordered in a set. RNNs can only be applied on sets by assuming an order in the data, which however biases the model towards a non-existing order in the data. Vinyals et al. (2015) and other papers have shown that the assumed order can have a significant impact on the model's performance, and hence, we should try to not use RNNs on sets. Ideally, our model should be permutation-equivariant/invariant such that the output is the same no matter how we sort the elements in a set.

Transformers offer the perfect architecture for this as the Multi-Head Attention is permutation-equivariant, and thus, outputs the same values no matter in what order we enter the inputs (inputs and outputs are permuted equally). The task we are looking at for sets is *Set Anomaly Detection* which means that we try to find the element(s) in a set that does not fit the others. In the research community, the common application of anomaly detection is performed on a set of images, where $N - 1$ images belong to the same category/have the same high-level features while one belongs to another category. Note that category does not necessarily have to relate to a class in a standard classification problem, but could be the combination of multiple features. For instance, on a face dataset, this could be people with glasses, male, beard, etc. An example of distinguishing different animals can be seen below. The first four images show foxes, while the last represents a different animal. We want to recognize that the last image shows a different animal, but it is not relevant which class of animal it is.



In this tutorial, we will use the CIFAR100 dataset. CIFAR100 has 600 images for 100 classes each with a resolution of 32x32, similar to CIFAR10. The larger amount of classes requires the model to attend to specific features in the images instead of coarse features as in CIFAR10, therefore making the task harder. We will show the model a set of 9 images of one class, and 1 image from another class. The task is to find the image that is from a different class than the other images. Using the raw images directly as input to the Transformer is not a good idea, because it is not translation invariant as a CNN, and would need to learn to detect image features from high-dimensional input first of all. Instead, we will use a pre-trained ResNet34 model from the package `flaxmodels` ([link](#)) to obtain high-level, low-dimensional features of the images. The ResNet model has been pre-trained on the `ImageNet` dataset which contains 1 million images of 1k classes and varying resolutions. However, during training and testing, the images are usually scaled to a resolution of 224x224, and hence we rescale our CIFAR images to this resolution as well. Below, we will load the dataset, and prepare the data for being processed by the ResNet model.

```
[30]: def image_to_numpy(img):
    img = np.array(img, dtype=np.float32)
    img = img / 255. # Normalization is done in the ResNet
    return img

# Resize to 224x224, and map to JAX
transform = transforms.Compose([transforms.Resize((224,224)),
                               image_to_numpy
                             ])
# Loading the training dataset.
train_set = CIFAR100(root=DATASET_PATH, train=True, transform=transform, download=True)

# Loading the test set
test_set = CIFAR100(root=DATASET_PATH, train=False, transform=transform, download=True)
```

(continues on next page)

(continued from previous page)

```
# For later, keep a dictionary mapping class indices to class names
class_idx_to_name = {val: key for key, val in train_set.class_to_idx.items()}

Files already downloaded and verified
Files already downloaded and verified
```

Next, we want to run the pre-trained ResNet model on the images, and extract the features before the classification layer. These are the most high-level features, and should sufficiently describe the images. CIFAR100 has some similarity to ImageNet, and thus we are not retraining the ResNet model in any form. However, if you would want to get the best performance and have a very large dataset, it would be better to add the ResNet to the computation graph during training and finetune its parameters as well. As we don't have a large enough dataset and want to train our model efficiently, we will extract the features beforehand. Let's load and prepare the model below.

```
[31]: # Import and install flaxmodels if needed
try:
    import flaxmodels
except ModuleNotFoundError:
    !pip install --upgrade git+https://github.com/matthias-wright/flaxmodels.git
    import flaxmodels

# Pretrained ResNet34 on ImageNet
resnet34 = flaxmodels.ResNet34(output='activations', pretrained='imagenet',
                                normalize=True)
main_rng, resnet_rng = random.split(main_rng, 2)
resnet_params = resnet34.init(resnet_rng, jnp.zeros((1, 224, 224, 3)))
# Jit its forward pass for efficiency
apply_resnet = jax.jit(lambda imgs: resnet34.apply(resnet_params, imgs, train=False))
```

We will now write a extraction function for the features below. This cell requires access to a GPU, as the model is rather deep and the images relatively large. The GPUs on GoogleColab are sufficient, but running this cell can take 2-3 minutes. Once it is run, the features are exported on disk so they don't have to be recalculated every time you run the notebook. However, this requires >150MB free disk space. So it is recommended to run this only on a local computer if you have enough free disk and a GPU (GoogleColab is fine for this). If you do not have a GPU, you can download the features from the [GoogleDrive folder](#).

```
[32]: def extract_features(dataset, save_file):
    if not os.path.isfile(save_file):
        data_loader = data.DataLoader(dataset, batch_size=128, shuffle=False, drop_
        last=False,
                                         collate_fn=lambda batch: np.stack([b[0] for b in_
        batch], axis=0))
        extracted_features = []
        for imgs in tqdm(data_loader):
            feats = apply_resnet(imgs)
            # Average pooling on the last conv features to obtain a image-level feature_
            vector
            feats = feats['block4_2'].mean(axis=(1,2))
            extracted_features.append(feats)
        extracted_features = jnp.concatenate(extracted_features, axis=0)
        extracted_features = jax.device_get(extracted_features)
        np.savez_compressed(save_file, feats=extracted_features)
    else:
```

(continues on next page)

(continued from previous page)

```

    extracted_features = np.load(save_file)['feats']
    return extracted_features

train_feat_file = os.path.join(CHECKPOINT_PATH, "train_set_features.npz")
train_set_feats = extract_features(train_set, train_feat_file)

test_feat_file = os.path.join(CHECKPOINT_PATH, "test_set_features.npz")
test_feats = extract_features(test_set, test_feat_file)

```

Let's verify the feature shapes below. The training should have 50k elements, and the test 10k images. The feature dimension is 512 for the ResNet34. If you experiment with other models, you likely see a different feature dimension.

[33]:

```

print("Train:", train_set_feats.shape)
print("Test: ", test_feats.shape)

Train: (50000, 512)
Test:  (10000, 512)

```

As usual, we want to create a validation set to detect when we should stop training. In this case, we will split the training set into 90% training, 10% validation. However, the difficulty is here that we need to ensure that the validation set has the same number of images for all 100 labels. Otherwise, we have a class imbalance which is not good for creating the image sets. Hence, we take 10% of the images for each class, and move them into the validation set. The code below does exactly this.

[34]:

```

## Split train into train+val
# Get labels from train set
labels = np.array(train_set.targets, dtype=np.int32)

# Get indices of images per class
num_labels = labels.max()+1
sorted_indices = np.argsort(labels).reshape(num_labels, -1) # [classes, num_imgs_per_
→class]

# Determine number of validation images per class
num_val_exmps = sorted_indices.shape[1] // 10

# Get image indices for validation and training
val_indices = sorted_indices[:, :num_val_exmps].reshape(-1)
train_indices = sorted_indices[:, num_val_exmps: ].reshape(-1)

# Group corresponding image features and labels
train_feats, train_labels = train_set_feats[train_indices], labels[train_indices]
val_feats, val_labels = train_set_feats[val_indices], labels[val_indices]

```

Now we can prepare a dataset class for the set anomaly task. We define an epoch to be the sequence in which each image has been exactly once as an “anomaly”. Hence, the length of the dataset is the number of images in it. For the training set, each time we access an item with `__getitem__`, we sample a random, different class than the image at the corresponding index `idx` has. In a second step, we sample $N - 1$ images of this sampled class. The set of 10 images is finally returned. The randomness in the `__getitem__` allows us to see a slightly different set during each iteration. However, we can't use the same strategy for the test set as we want the test dataset to be the same every time we iterate over it. Hence, we sample the sets in the `__init__` method, and return those in `__getitem__`. The code below implements exactly this dynamic.

```
[35]: class SetAnomalyDataset(data.Dataset):

    def __init__(self, img_feats, labels, np_rng, set_size=10, train=True):
        """
        Inputs:
            img_feats - Tensor of shape [num_imgs, img_dim]. Represents the high-level
            features.
            labels - Tensor of shape [num_imgs], containing the class labels for the
            images
            set_size - Number of elements in a set. N-1 are sampled from one class, and
            one from another one.
            train - If True, a new set will be sampled every time __getitem__ is called.
        """
        super().__init__()
        self.img_feats = img_feats
        self.labels = labels
        self.np_rng = np_rng
        self.set_size = set_size # The set size is here the number of images from the
        same class per set
        self.train = train

        # Tensors with indices of the images per class
        self.num_labels = labels.max()+1
        self.img_idx_by_label = np.argsort(self.labels).reshape(self.num_labels, -1)

        if not train:
            self.test_sets = self._create_test_sets()

    def _create_test_sets(self):
        # Pre-generates the sets for each image for the test set
        test_sets = []
        num_imgs = self.img_feats.shape[0]
        test_sets = [self.sample_img_set(self.labels[idx]) for idx in range(num_imgs)]
        test_sets = np.stack(test_sets, axis=0)
        return test_sets

    def sample_img_set(self, anomaly_label):
        """
        Samples a new set of images, given the label of the anomaly.
        The sampled images come from a different class than anomaly_label
        """
        # Sample class from 0,...,num_classes-1 while skipping anomaly_label as class
        set_label = self.np_rng.integers(self.num_labels-1)
        if set_label >= anomaly_label:
            set_label += 1

        # Sample images from the class determined above
        img_indices = self.np_rng.choice(self.img_idx_by_label.shape[1], size=self.set_
        size, replace=False)
        img_indices = self.img_idx_by_label[set_label, img_indices]
        return img_indices

    def __len__(self):
```

(continues on next page)

(continued from previous page)

```

    return self.img_feats.shape[0]

def __getitem__(self, idx):
    anomaly = self.img_feats[idx]
    if self.train: # If train => sample
        img_indices = self.sample_img_set(self.labels[idx])
    else: # If test => use pre-generated ones
        img_indices = self.test_sets[idx]

    # Concatenate images. The anomaly is always the last image for simplicity
    img_set = np.concatenate([self.img_feats[img_indices], anomaly[None]], axis=0)
    indices = np.concatenate([img_indices, np.array([idx], dtype=np.int32)], axis=0)
    label = img_set.shape[0]-1

    # We return the indices of the images for visualization purpose. "Label" is the
    # index of the anomaly
    return img_set, indices, label

```

Next, we can setup our datasets and data loaders below. Here, we will use a set size of 10, i.e. 9 images from one category + 1 anomaly. Feel free to change it if you want to experiment with the sizes.

```
[36]: SET_SIZE = 10
test_labels = np.array(test_set.targets, dtype=np.int32)

anom_train_dataset = SetAnomalyDataset(train_feats, train_labels, np_rng=np.random.
    default_rng(42), set_size=SET_SIZE, train=True)
anom_val_dataset = SetAnomalyDataset(val_feats, val_labels, np_rng=np.random.
    default_rng(43), set_size=SET_SIZE, train=False)
anom_test_dataset = SetAnomalyDataset(test_feats, test_labels, np_rng=np.random.
    default_rng(123), set_size=SET_SIZE, train=False)

anom_train_loader = data.DataLoader(anom_train_dataset, batch_size=64, shuffle=True,
    drop_last=True, collate_fn= numpy_collate)
anom_val_loader = data.DataLoader(anom_val_dataset, batch_size=64, shuffle=False,
    drop_last=False, collate_fn= numpy_collate)
anom_test_loader = data.DataLoader(anom_test_dataset, batch_size=64, shuffle=False,
    drop_last=False, collate_fn= numpy_collate)
```

To understand the dataset a little better, we can plot below a few sets from the test dataset. Each row shows a different input set, where the first 9 are from the same class.

```
[37]: def visualize_exmp(indices, orig_dataset):
    images = [orig_dataset[idx][0] for idx in indices.reshape(-1)]
    images = jax.device_get(jnp.stack(images, axis=0)).astype(np.float32)
    images = torch.from_numpy(images)
    images = images.permute(0, 3, 1, 2)
    img_grid = torchvision.utils.make_grid(images, nrow=SET_SIZE, normalize=True, pad_
    value=0.5, padding=16)
    img_grid = img_grid.permute(1, 2, 0)

    plt.figure(figsize=(12,8))
    plt.title("Anomaly examples on CIFAR100")
```

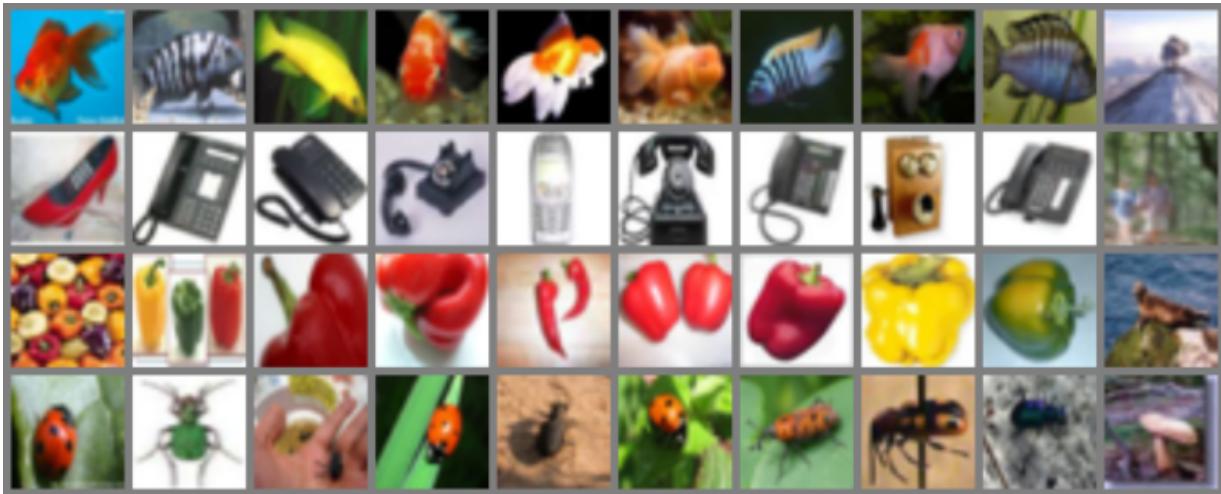
(continues on next page)

(continued from previous page)

```
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()

_, indices, _ = next(iter(anom_test_loader))
visualize_exmp(indices[:4], test_set)
```

Anomaly examples on CIFAR100



We can already see that for some sets the task might be easier than for others. Difficulties can especially arise if the anomaly is in a different, but yet visually similar class (e.g. insect vs mushroom, train vs bus, etc.).

After having prepared the data, we can look closer at the model. Here, we have a classification of the whole set. For the prediction to be permutation-equivariant, we will output one logit for each image. Over these logits, we apply a softmax and train the anomaly image to have the highest score/probability. This is a bit different than a standard classification layer as the softmax is applied over images, not over output classes in the classical sense. However, if we swap two images in their position, we effectively swap their position in the output softmax. Hence, the prediction is equivariant with respect to the input. Furthermore, we need to remove the positional encoding since these features would break the permutation equivariance. We implement this setup below in the subclass of the Trainer module.

(continues on next page)

(continued from previous page)

```

logits = logits.squeeze(axis=-1)
loss = optax.softmax_cross_entropy_with_integer_labels(logits, labels).mean()
acc = (logits.argmax(axis=-1) == labels).astype(jnp.float32).mean()
return loss, (acc, rng)
return calculate_loss

```

Finally, we write our train function below. It has the exact same structure as the reverse task one, hence not much of an explanation is needed here.

```
[39]: def train_anomaly(max_epochs=100, **model_args):
    num_train_iters = len(anom_train_loader) * max_epochs
    # Create a trainer module with specified hyperparameters
    trainer = AnomalyTrainer(model_name='SetAnomalyTask',
        example_batch=next(iter(anom_train_loader)),
        max_iters=num_train_iters,
        **model_args)
    if not trainer.checkpoint_exists(): # Skip training if pretrained model exists
        trainer.train_model(anom_train_loader, anom_val_loader, num_epochs=max_epochs)
        trainer.load_model()
    else:
        trainer.load_model(pretrained=True)
    train_acc = trainer.eval_model(anom_train_loader)
    val_acc = trainer.eval_model(anom_val_loader)
    test_acc = trainer.eval_model(anom_test_loader)
    # Bind parameters to model for easier inference
    trainer.model_bd = trainer.model.bind({'params': trainer.state.params})
    return trainer, {'train_acc': train_acc, 'val_acc': val_acc, 'test_acc': test_acc}
```

Let's finally train our model. We will use 4 layers with 4 attention heads each. The hidden dimensionality of the model is 256, and we use a dropout of 0.1 throughout the model for good regularization. Note that we also apply the dropout on the input features, as this makes the model more robust against image noise and generalizes better. Again, we use warmup to slowly start our model training.

```
[40]: anomaly_trainer, anomaly_result = train_anomaly(model_dim=256,
                                                    num_heads=4,
                                                    num_classes=1,
                                                    num_layers=4,
                                                    dropout_prob=0.1,
                                                    input_dropout_prob=0.1,
                                                    lr=5e-4,
                                                    warmup=100)
```

We can print the achieved accuracy below.

```
[41]: print(f"Train accuracy: {(100.0*anomaly_result['train_acc']):.2f}%")
print(f"Val accuracy:   {(100.0*anomaly_result['val_acc']):.2f}%")
print(f"Test accuracy:  {(100.0*anomaly_result['test_acc']):.2f}%")

Train accuracy: 98.37%
Val accuracy:   94.50%
Test accuracy:  94.66%
```

With ~94% validation and test accuracy, the model generalizes quite well. It should be noted that you might see slightly different scores depending on what computer/device you are running this notebook. This is because despite

setting the seed before generating the test dataset, it may not always be the same across platforms and numpy versions. Nevertheless, we can conclude that the model performs quite well and can solve the task for most sets. Before trying to interpret the model, let's verify that our model is permutation-equivariant, and assigns the same predictions for different permutations of the input set. For this, we sample a batch from the test set and run it through the model to obtain the probabilities.

```
[42]: inp_data, indices, labels = next(iter(anom_test_loader))
preds = anomaly_trainer.model_bd(inp_data, add_positional_encoding=False, train=False)
preds = jax.nn.softmax(preds.squeeze(axis=-1))

permut = np.random.permutation(inp_data.shape[1])
permut_inp_data = inp_data[:,permut]
perm_preds = anomaly_trainer.model_bd(permut_inp_data, add_positional_encoding=False, ↴
                                         train=False)
perm_preds = jax.nn.softmax(perm_preds.squeeze(axis=-1))

preds = jax.device_get(preds)
perm_preds = jax.device_get(perm_preds)

assert np.abs(preds[:,permut] - perm_preds).max() < 1e-5, "Predictions are not ↴
                                         permutation equivariant"

print("Preds\n", preds[0,permut])
print("Permuted preds\n", perm_preds[0])

Preds
[1.3902171e-07 4.3522828e-08 5.2554757e-08 1.2441276e-08 2.7709259e-08
 9.9999952e-01 5.5640967e-08 3.5155960e-08 1.4563368e-08 4.8264688e-08]
Permuted preds
[1.3902175e-07 4.3522839e-08 5.2554768e-08 1.2441279e-08 2.7709264e-08
 9.9999976e-01 5.5640978e-08 3.5155971e-08 1.4563372e-08 4.8264699e-08]
```

You can see that the predictions are almost exactly the same, and only differ because of slight numerical differences inside the network operation.

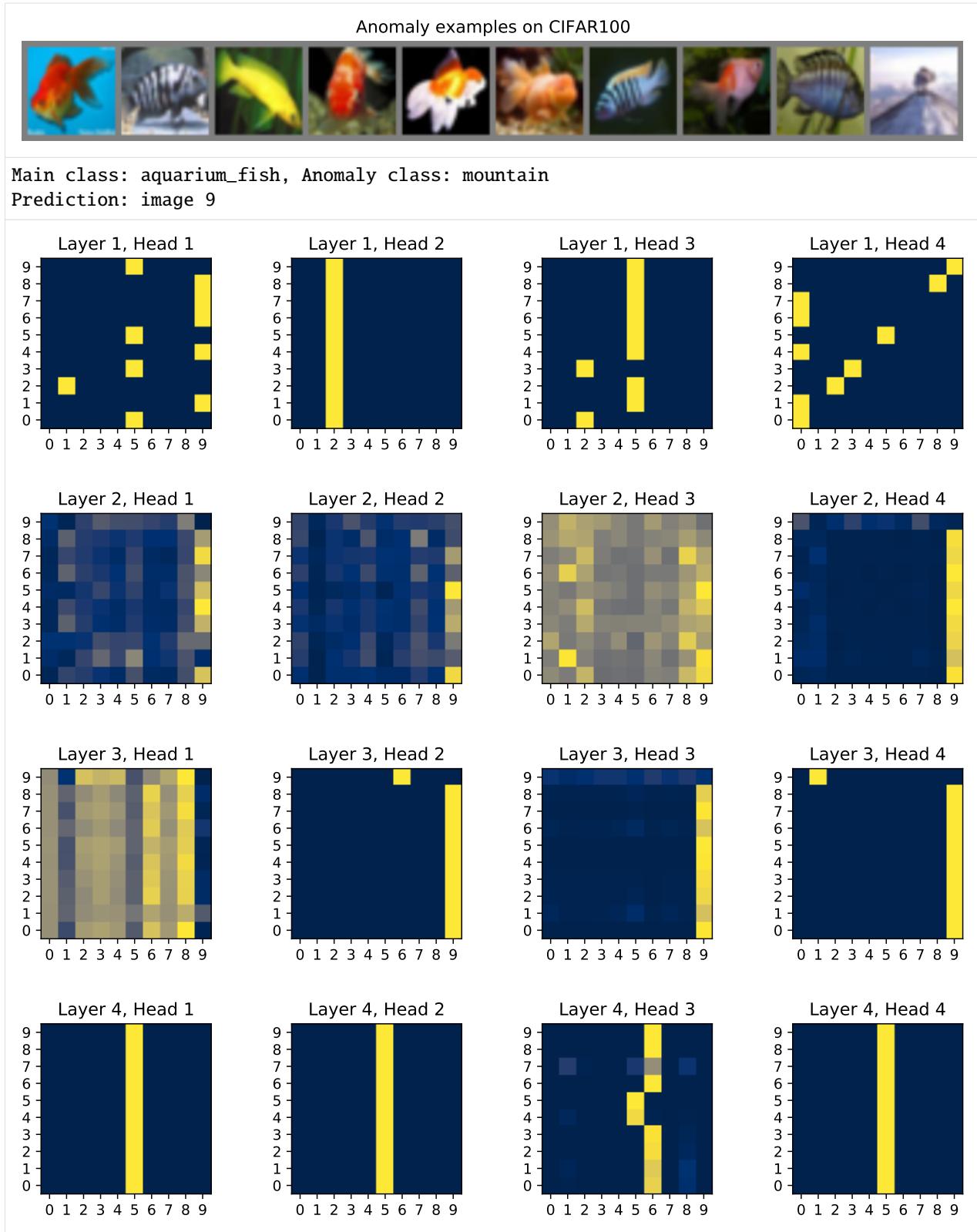
To interpret the model a little more, we can plot the attention maps inside the model. This will give us an idea of what information the model is sharing/communicating between images, and what each head might represent. First, we need to extract the attention maps for the test batch above, and determine the discrete predictions for simplicity.

```
[43]: attention_maps = anomaly_trainer.model_bd.get_attention_maps(inp_data, add_positional_ ↴
                                                               encoding=False, train=False)
predictions = preds.argmax(axis=-1)
```

Below we write a plot function which plots the images in the input set, the prediction of the model, and the attention maps of the different heads on layers of the transformer. Feel free to explore the attention maps for different input examples as well.

```
[44]: def visualize_prediction(idx):
    visualize_exmp(indices[idx:idx+1], test_set)
    print(f'Main class: {class_idx_to_name[test_labels[indices[idx, 0]]]}, Anomaly class: ↴
          {class_idx_to_name[test_labels[indices[idx, -1]]]}')
    print(f'Prediction: image {predictions[idx].item()}')
    plot_attention_maps(input_data=None, attn_maps=attention_maps, idx=idx)

visualize_prediction(0)
```



Depending on the random seed, you might see a slightly different input set. For the version on the website, we compare 9 aquarium fish images with a volcano/mountain. We see that multiple heads, for instance, Layer 2 Head 1, Layer 2

Head 4, and Layer 3 Head 2-4 focus on the last image. Additionally, the heads in Layer 4 all seem to ignore the last image and assign a very low attention probability to it. This shows that the model has indeed recognized that the image doesn't fit the setting, and hence predicted it to be the anomaly. Layer 2 Head 3 and Layer 3 Head 1 seems to take a slightly weighted average of all images. That might indicate that the model extracts the “average” information of all images, to compare it to the image features itself.

Let's try to find where the model actually makes a mistake. We can do this by identifying the sets where the model predicts something else than 9, as in the dataset, we ensured that the anomaly is always at the last position in the set.

```
[45]: mistakes = np.where(predictions != 9)[0]
print("Indices with mistake:", mistakes)

Indices with mistake: [10 58]
```

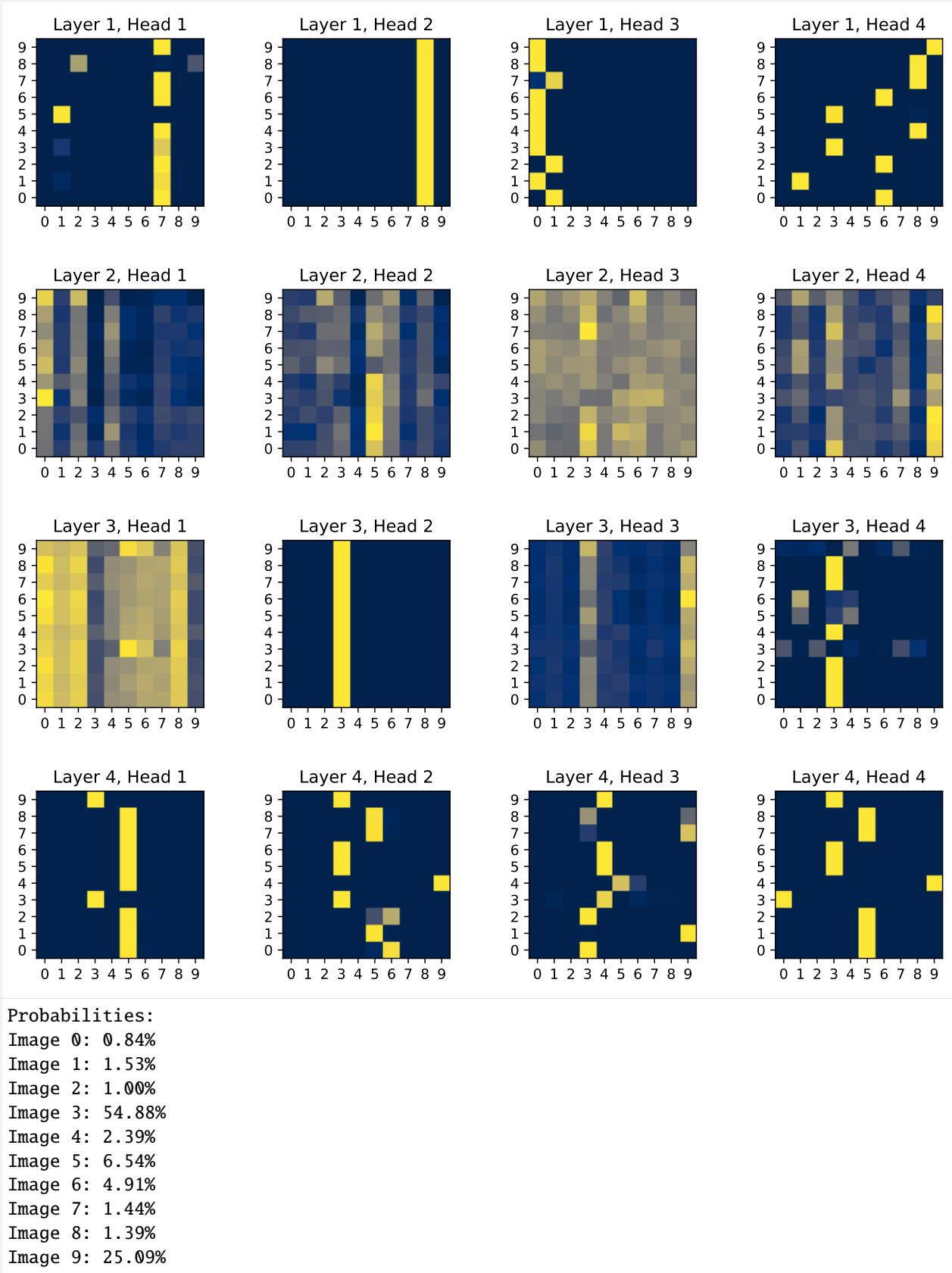
As our model achieves ~94% accuracy, we only have very little number of mistakes in a batch of 64 sets. Still, let's visualize one of them, for example the last one:

```
[46]: visualize_prediction(mistakes[0])
print("Probabilities:")
for i, p in enumerate(preds[mistakes[0]]):
    print(f"Image {i}: {100.0*p:.2f}%")
```

Anomaly examples on CIFAR100



Main class: forest, Anomaly class: sea
Prediction: image 3



In this example, the model confuses a picture of a sea with a forest, giving a probability of ~55% to image 3, and 25% to the actual anomaly. However, the difficulty here is that the picture of the sea actually contains trees in the foreground, which makes the image a bit of an ambiguous class. It is possible a picture of a sea taken out of a forest, which confuses the model. Nevertheless, in general, the model performs quite well.

4.23.3 Conclusion

In this tutorial, we took a closer look at the Multi-Head Attention layer which uses a scaled dot product between queries and keys to find correlations and similarities between input elements. The Transformer architecture is based on the Multi-Head Attention layer and applies multiple of them in a ResNet-like block. The Transformer is a very important, recent architecture that can be applied to many tasks and datasets. Although it is best known for its success in NLP, there is so much more to it. We have seen its application on sequence-to-sequence tasks and set anomaly detection. Its property of being permutation-equivariant if we do not provide any positional encodings, allows it to generalize to many settings. Hence, it is important to know the architecture, but also its possible issues such as the gradient problem during the first iterations solved by learning rate warm-up. If you are interested in continuing with the study of the Transformer architecture, please have a look at the blog posts listed at the beginning of the tutorial notebook.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.24 Tutorial 7 (JAX): Graph Neural Networks

Filled notebook:

Pre-trained models:

PyTorch version:

Author: Phillip Lippe

Note: This notebook is written in JAX+Flax. It is a 1-to-1 translation of the original notebook written in PyTorch+PyTorch Lightning with almost identical results. For an introduction to JAX, check out our [Tutorial 2 \(JAX\): Introduction to JAX+Flax](#). Further, throughout the notebook, we comment on major differences to the PyTorch version and provide explanations for the major parts of the JAX code.

In this tutorial, we will discuss the application of neural networks on graphs. Graph Neural Networks (GNNs) have recently gained increasing popularity in both applications and research, including domains such as social networks, knowledge graphs, recommender systems, and bioinformatics. While the theory and math behind GNNs might first seem complicated, the implementation of those models is quite simple and helps in understanding the methodology. Therefore, we will discuss the implementation of basic network layers of a GNN, namely graph convolutions, and attention layers.

Below, we will start by importing our standard libraries. We will use JAX, Flax and Optax for training our models.

```
[1]: ## Standard libraries
import os
import json
import math
import numpy as np
import time

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()
sns.set()

## Progress bar
from tqdm.notebook import tqdm

## To run JAX on TPU in Google Colab, uncomment the two lines below
# import jax.tools.colab_tpu
# jax.tools.colab_tpu.setup_tpu()

## JAX
import jax
import jax.numpy as jnp
from jax import random

## Flax (NN in JAX)
try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax
from flax import linen as nn
from flax.training import train_state, checkpoints

## Optax (Optimizers in JAX)
try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax

## PyTorch
import torch
import torch.utils.data as data
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms
```

(continues on next page)

(continued from previous page)

```
# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../../../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../../../saved_models/tutorial7_jax"

# Seeding for random operations
main_rng = random.PRNGKey(42)

print("Device:", jax.devices()[0])

/tmp/ipykernel_2384472/156319451.py:12: DeprecationWarning: `set_matplotlib_formats` is
  deprecated since IPython 7.23, directly use `matplotlib_inline.backend_inline.set_
  matplotlib_formats()`
    set_matplotlib_formats('svg', 'pdf') # For export
Device: gpu:0
```

We also have a few pre-trained models we download below.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/JAX/tutorial7/"
# Files to download
pretrained_files = []

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please contact the author with the full output
      including the following error:\n", e)
```

4.24.1 Graph Neural Networks

Graph representation

Before starting the discussion of specific neural network operations on graphs, we should consider how to represent a graph. Mathematically, a graph \mathcal{G} is defined as a tuple of a set of nodes/vertices V , and a set of edges/links E : $\mathcal{G} = (V, E)$. Each edge is a pair of two vertices, and represents a connection between them. For instance, let's look at the following graph:

The vertices are $V = \{1, 2, 3, 4\}$, and edges $E = \{(1, 2), (2, 3), (2, 4), (3, 4)\}$. Note that for simplicity, we assume the graph to be undirected and hence don't add mirrored pairs like $(2, 1)$. In application, vertices and edge can often have specific attributes, and edges can even be directed. The question is how we could represent this diversity in an efficient way for matrix operations. Usually, for the edges, we decide between two variants: an adjacency matrix, or a list of paired vertex indices.

The **adjacency matrix** A is a square matrix whose elements indicate whether pairs of vertices are adjacent, i.e. connected, or not. In the simplest case, A_{ij} is 1 if there is a connection from node i to j , and otherwise 0. If we have edge attributes or different categories of edges in a graph, this information can be added to the matrix as well. For an undirected graph, keep in mind that A is a symmetric matrix ($A_{ij} = A_{ji}$). For the example graph above, we have the following adjacency matrix:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

While expressing a graph as a list of edges is more efficient in terms of memory and (possibly) computation, using an adjacency matrix is more intuitive and simpler to implement. In our implementations below, we will rely on the adjacency matrix to keep the code simple. However, common libraries use edge lists, which we will discuss later more.

Graph Convolutions

Graph Convolutional Networks have been introduced by Kipf et al. in 2016 at the University of Amsterdam. He also wrote a great [blog post](#) about this topic, which is recommended if you want to read about GCNs from a different perspective. GCNs are similar to convolutions in images in the sense that the “filter” parameters are typically shared over all locations in the graph. At the same time, GCNs rely on message passing methods, which means that vertices exchange information with the neighbors, and send “messages” to each other. Before looking at the math, we can try to visually understand how GCNs work. The first step is that each node creates a feature vector that represents the message it wants to send to all its neighbors. In the second step, the messages are sent to the neighbors, so that a node receives one message per adjacent node. Below we have visualized the two steps for our example graph.

If we want to formulate that in more mathematical terms, we need to first decide how to combine all the messages a node receives. As the number of messages vary across nodes, we need an operation that works for any number. Hence, the usual way to go is to sum or take the mean. Given the previous features of nodes $H^{(l)}$, the GCN layer is defined as follows:

$$H^{(l+1)} = \sigma \left(\hat{D}^{-1/2} \hat{A} \hat{D}^{-1/2} H^{(l)} W^{(l)} \right)$$

$W^{(l)}$ is the weight parameters with which we transform the input features into messages ($H^{(l)} W^{(l)}$). To the adjacency matrix A we add the identity matrix so that each node sends its own message also to itself: $\hat{A} = A + I$. Finally, to take the average instead of summing, we calculate the matrix \hat{D} which is a diagonal matrix with D_{ii} denoting the number of neighbors node i has. σ represents an arbitrary activation function, and not necessarily the sigmoid (usually a ReLU-based activation function is used in GNNs).

When implementing the GCN layer in PyTorch, we can take advantage of the flexible operations on tensors. Instead of defining a matrix \hat{D} , we can simply divide the summed messages by the number of neighbors afterward. Additionally, we replace the weight matrix with a linear layer, which additionally allows us to add a bias. Written as a PyTorch module, the GCN layer is defined as follows:

```
[3]: class GCNLayer(nn.Module):
    c_out : int # Output feature size

    @nn.compact
    def __call__(self, node_feats, adj_matrix):
        """
        Inputs:
            node_feats - Array with node features of shape [batch_size, num_nodes, c_in]
            adj_matrix - Batch of adjacency matrices of the graph. If there is an edge
            ↪from i to j, adj_matrix[b,i,j]=1 else 0.
                    Supports directed edges by non-symmetric matrices. Assumes to
            ↪already have added the identity connections.
            Shape: [batch_size, num_nodes, num_nodes]
        """
        # Num neighbours = number of incoming edges
        num_neighbours = adj_matrix.sum(axis=-1, keepdims=True)
        node_feats = nn.Dense(features=self.c_out, name='projection')(node_feats)
        node_feats = jax.lax.batch_matmul(adj_matrix, node_feats)
        node_feats = node_feats / num_neighbours
        return node_feats
```

To further understand the GCN layer, we can apply it to our example graph above. First, let's specify some node features and the adjacency matrix with added self-connections:

```
[4]: node_feats = jnp.arange(8, dtype=jnp.float32).reshape((1, 4, 2))
adj_matrix = jnp.array([[[], 1, 0, 0],
                      [1, 1, 1, 1],
                      [0, 1, 1, 1],
                      [0, 1, 1, 1]]].astype(jnp.float32)

print("Node features:\n", node_feats)
print("\nAdjacency matrix:\n", adj_matrix)

Node features:
[[[0. 1.]
  [2. 3.]
  [4. 5.]
  [6. 7.]]]

Adjacency matrix:
[[[1. 1. 0. 0.]
  [1. 1. 1. 1.]
  [0. 1. 1. 1.]
  [0. 1. 1. 1.]]]
```

Next, let's apply a GCN layer to it. For simplicity, we initialize the linear weight matrix as an identity matrix so that the input features are equal to the messages. This makes it easier for us to verify the message passing operation.

```
[5]: layer = GCNLayer(c_out=2)
```

(continues on next page)

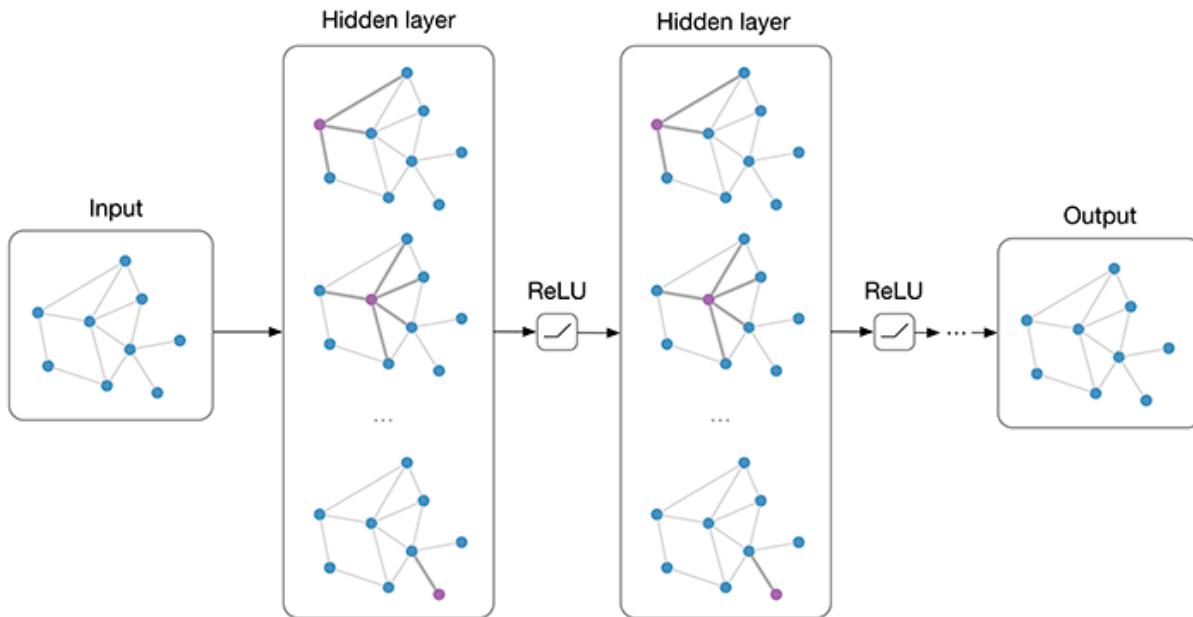
(continued from previous page)

```
# We define our own parameters here instead of using random initialization
params = {'projection': {
    'kernel': jnp.array([[1., 0.], [0., 1.]]),
    'bias': jnp.array([0., 0.])
}}
out_feats = layer.apply({'params': params}, node_feats, adj_matrix)

print("Adjacency matrix", adj_matrix)
print("Input features", node_feats)
print("Output features", out_feats)
```

Adjacency matrix [[1. 1. 0. 0.]
[1. 1. 1. 1.]
[0. 1. 1. 1.]
[0. 1. 1. 1.]]]
Input features [[[0. 1.]
[2. 3.]
[4. 5.]
[6. 7.]]]]
Output features [[[1. 2.]
[3. 4.]
[4. 5.]
[4. 5.]]]]

As we can see, the first node's output values are the average of itself and the second node. Similarly, we can verify all other nodes. However, in a GNN, we would also want to allow feature exchange between nodes beyond its neighbors. This can be achieved by applying multiple GCN layers, which gives us the final layout of a GNN. The GNN can be built up by a sequence of GCN layers and non-linearities such as ReLU. For a visualization, see below (figure credit - Thomas Kipf, 2016).



However, one issue we can see from looking at the example above is that the output features for nodes 3 and 4 are the same because they have the same adjacent nodes (including itself). Therefore, GCN layers can make the network forget node-specific information if we just take a mean over all messages. Multiple possible improvements have been

proposed. While the simplest option might be using residual connections, the more common approach is to either weigh the self-connections higher or define a separate weight matrix for the self-connections. Alternatively, we can re-visit a concept from the last tutorial: attention.

Graph Attention

If you remember from the last tutorial, attention describes a weighted average of multiple elements with the weights dynamically computed based on an input query and elements' keys (if you haven't read Tutorial 6 yet, it is recommended to at least go through the very first section called [What is Attention?](#)). This concept can be similarly applied to graphs, one of such is the Graph Attention Network (called GAT, proposed by [Velickovic et al., 2017](#)). Similarly to the GCN, the graph attention layer creates a message for each node using a linear layer/weight matrix. For the attention part, it uses the message from the node itself as a query, and the messages to average as both keys and values (note that this also includes the message to itself). The score function f_{attn} is implemented as a one-layer MLP which maps the query and key to a single value. The MLP looks as follows (figure credit - [Velickovic et al.](#)):

h_i and h_j are the original features from node i and j respectively, and represent the messages of the layer with \mathbf{W} as weight matrix. \mathbf{a} is the weight matrix of the MLP, which has the shape $[1, 2 \times d_{\text{message}}]$, and α_{ij} the final attention weight from node i to j . The calculation can be described as follows:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_j]))}{\sum_{k \in \mathcal{N}_i} \exp(\text{LeakyReLU}(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_k]))}$$

The operator $||$ represents the concatenation, and \mathcal{N}_i the indices of the neighbors of node i . Note that in contrast to usual practice, we apply a non-linearity (here LeakyReLU) before the softmax over elements. Although it seems like a minor change at first, it is crucial for the attention to depend on the original input. Specifically, let's remove the non-linearity for a second, and try to simplify the expression:

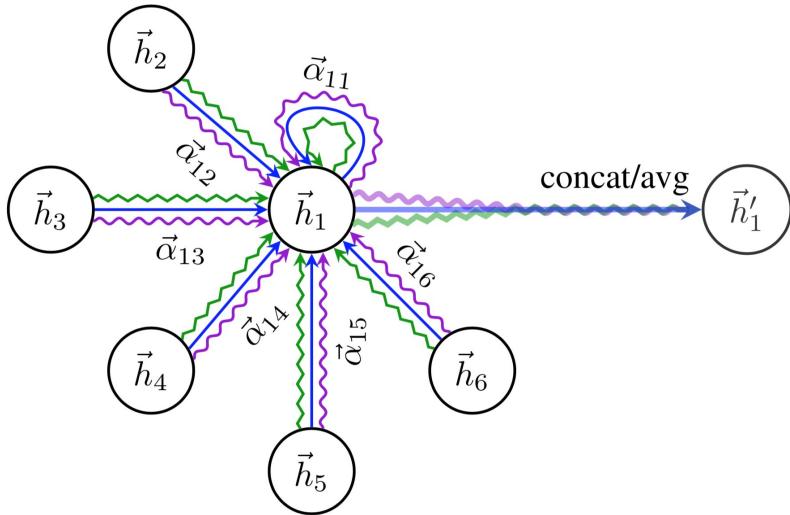
$$\begin{aligned} \alpha_{ij} &= \frac{\exp(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_j])}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_k])} \\ &= \frac{\exp(\mathbf{a}_{:,d/2} \mathbf{W}h_i + \mathbf{a}_{:,d/2} \mathbf{W}h_j)}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}_{:,d/2} \mathbf{W}h_i + \mathbf{a}_{:,d/2} \mathbf{W}h_k)} \\ &= \frac{\exp(\mathbf{a}_{:,d/2} \mathbf{W}h_i) \cdot \exp(\mathbf{a}_{:,d/2} \mathbf{W}h_j)}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}_{:,d/2} \mathbf{W}h_i) \cdot \exp(\mathbf{a}_{:,d/2} \mathbf{W}h_k)} \\ &= \frac{\exp(\mathbf{a}_{:,d/2} \mathbf{W}h_j)}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}_{:,d/2} \mathbf{W}h_k)} \end{aligned}$$

We can see that without the non-linearity, the attention term with h_i actually cancels itself out, resulting in the attention being independent of the node itself. Hence, we would have the same issue as the GCN of creating the same output features for nodes with the same neighbors. This is why the LeakyReLU is crucial and adds some dependency on h_i to the attention.

Once we obtain all attention factors, we can calculate the output features for each node by performing the weighted average:

$$h'_i = \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}h_j \right)$$

σ is yet another non-linearity, as in the GCN layer. Visually, we can represent the full message passing in an attention layer as follows (figure credit - [Velickovic et al.](#)):



To increase the expressiveness of the graph attention network, Velickovic et al. proposed to extend it to multiple heads similar to the Multi-Head Attention block in Transformers. This results in N attention layers being applied in parallel. In the image above, it is visualized as three different colors of arrows (green, blue, and purple) that are afterward concatenated. The average is only applied for the very final prediction layer in a network.

After having discussed the graph attention layer in detail, we can implement it below:

```
[6]: class GATLayer(nn.Module):
    c_out : int # Dimensionality of output features
    num_heads : int # Number of heads, i.e. attention mechanisms to apply in parallel.
    concat_heads : bool = True # If True, the output of the different heads is
    ↪ concatenated instead of averaged.
    alpha : float = 0.2 # Negative slope of the LeakyReLU activation.

    def setup(self):
        if self.concat_heads:
            assert self.c_out % self.num_heads == 0, "Number of output features must be
    ↪ a multiple of the count of heads."
            c_out_per_head = self.c_out // self.num_heads
        else:
            c_out_per_head = self.c_out

        # Sub-modules and parameters needed in the layer
        self.projection = nn.Dense(c_out_per_head * self.num_heads,
                                   kernel_init=nn.initializers.glorot_uniform())
        self.a = self.param('a',
                            nn.initializers.glorot_uniform(),
                            (self.num_heads, 2 * c_out_per_head)) # One per head

    def __call__(self, node_feats, adj_matrix, print_attn_probs=False):
        """
        Inputs:
            node_feats - Input features of the node. Shape: [batch_size, c_in]
            adj_matrix - Adjacency matrix including self-connections. Shape: [batch_size,
    ↪ num_nodes, num_nodes]
            print_attn_probs - If True, the attention weights are printed during the
    ↪ forward pass (for debugging purposes)
        """
        ...
```

(continues on next page)

(continued from previous page)

```

"""
batch_size, num_nodes = node_feats.shape[0], node_feats.shape[1]

# Apply linear layer and sort nodes by head
node_feats = self.projection(node_feats)
node_feats = node_feats.reshape((batch_size, num_nodes, self.num_heads, -1))

# We need to calculate the attention logits for every edge in the adjacency_
matrix
    # In order to take advantage of JAX's just-in-time compilation, we should not use
    # arrays with shapes that depend on e.g. the number of edges. Hence, we calculate
    # the logit for every possible combination of nodes. For efficiency, we can split
    #  $a[Wh_i // Wh_j] = a_{:d/2} * Wh_i + a_{d/2:} * Wh_j$ .
    logit_parent = (node_feats * self.a[None, None, :, :self.a.shape[0]//2]).sum(axis=-
1)
    logit_child = (node_feats * self.a[None, None, :, self.a.shape[0]//2:]).sum(axis=-1)
    attn_logits = logit_parent[:, :, None, :] + logit_child[:, None, :, :]
    attn_logits = nn.leaky_relu(attn_logits, self.alpha)

    # Mask out nodes that do not have an edge between them
    attn_logits = jnp.where(adj_matrix[..., None] == 1.,
                           attn_logits,
                           jnp.ones_like(attn_logits) * (-9e15))

    # Weighted average of attention
    attn_probs = nn.softmax(attn_logits, axis=2)
    if print_attn_probs:
        print("Attention probs\n", attn_probs.transpose(0, 3, 1, 2))
    node_feats = jnp.einsum('bijh,bjhc->bihc', attn_probs, node_feats)

    # If heads should be concatenated, we can do this by reshaping. Otherwise, take_
mean
    if self.concat_heads:
        node_feats = node_feats.reshape(batch_size, num_nodes, -1)
    else:
        node_feats = node_feats.mean(axis=2)

return node_feats

```

Again, we can apply the graph attention layer on our example graph above to understand the dynamics better. As before, the input layer is initialized as an identity matrix, but we set a to be a vector of arbitrary numbers to obtain different attention values. We use two heads to show the parallel, independent attention mechanisms working in the layer.

```
[7]: layer = GATLayer(2, num_heads=2)
params = {
    'projection': {
        'kernel': jnp.array([[1., 0.], [0., 1.]]),
        'bias': jnp.array([0., 0.])
    },
    'a': jnp.array([-0.2, 0.3], [0.1, -0.1])
}
out_feats = layer.apply({'params': params}, node_feats, adj_matrix, print_attn_
probs=True)
```

(continues on next page)

(continued from previous page)

```

print("Adjacency matrix", adj_matrix)
print("Input features", node_feats)
print("Output features", out_feats)

Attention probs
[[[0.35434368 0.6456563 0. 0. ]
 [0.10956533 0.14496915 0.26415104 0.48131457]
 [0. 0.18580717 0.2885041 0.52568877]
 [0. 0.23912403 0.2696116 0.49126437]]]

[[0.5099987 0.49000138 0. 0. ]
 [0.2975179 0.24358706 0.23403588 0.2248592 ]
 [0. 0.38382432 0.31424877 0.3019269 ]
 [0. 0.40175956 0.3289329 0.26930752]]]

Adjacency matrix [[1. 1. 0. 0.]
 [1. 1. 1. 1.]
 [0. 1. 1. 1.]
 [0. 1. 1. 1.]]
Input features [[[0. 1.]
 [2. 3.]
 [4. 5.]
 [6. 7.]]]

Output features [[[1.2913126 1.9800028]
 [4.23443 3.7724729]
 [4.6797633 4.8362055]
 [4.504281 4.735096 ]]]

```

We recommend that you try to calculate the attention matrix at least for one head and one node for yourself. The entries are 0 where there does not exist an edge between i and j . For the others, we see a diverse set of attention probabilities. Moreover, the output features of node 3 and 4 are now different although they have the same neighbors.

4.24.2 Conclusion

In this tutorial, we have seen the application of neural networks to graph structures. We looked at how a graph can be represented (adjacency matrix or edge list), and discussed the implementation of common graph layers: GCN and GAT. The implementations showed the practical side of the layers, which is often easier than the theory. For implementing full GNNs in JAX, we recommend taking a look at [Jraph](#). There are a lot of applications that benefit from GNNs, and the importance of these networks will likely increase over the next years.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.25 Tutorial 9 (JAX): Deep Autoencoders

Filled notebook:

Pre-trained models:

PyTorch version:

Author: Phillip Lippe

Note: This notebook is written in JAX+Flax. It is a 1-to-1 translation of the original notebook written in PyTorch+PyTorch Lightning with almost identical results. For an introduction to JAX, check out our [Tutorial 2 \(JAX\): Introduction to JAX+Flax](#). Further, throughout the notebook, we comment on major differences to the PyTorch version and provide explanations for the major parts of the JAX code.

Speed comparison: We note the training times for all models in the PyTorch and the JAX implementation below (PyTorch v1.11, JAX v0.3.13). The models were trained on the same hardware (NVIDIA RTX3090, 24 core CPU) and we slightly adjusted the tutorials to use the exact same training settings (same data loading parameters, evaluation schedule, etc.). Overall, the JAX implementation is about *1.8x faster* than PyTorch!

Models	PyTorch	JAX
AE - 64 latents	13min 10sec	7min 10sec
AE - 128 latents	13min 11sec	7min 10sec
AE - 256 latents	13min 11sec	7min 11sec
AE - 384 latents	13min 12sec	7min 14sec

In this tutorial, we will take a closer look at autoencoders (AE). Autoencoders are trained on encoding input data such as images into a smaller feature vector, and afterward, reconstruct it by a second neural network, called a decoder. The feature vector is called the “bottleneck” of the network as we aim to compress the input data into a smaller amount of features. This property is useful in many applications, in particular in compressing data or comparing images on a metric beyond pixel-level comparisons. Besides learning about the autoencoder framework, we will also see the “deconvolution” (or transposed convolution) operator in action for scaling up feature maps in height and width. Such deconvolution networks are necessary wherever we start from a small feature vector and need to output an image of full size (e.g. in VAE, GANs, or super-resolution applications).

First of all, we import most of our standard libraries. We use **JAX** as acceleration backend, **Flax** for implementing neural networks, and **Optax** to optimize the models.

```
[1]: ## Standard libraries
import os
import json
import math
import numpy as np
from scipy import spatial

## Imports for plotting
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()
sns.set()

## Progress bar
from tqdm.auto import tqdm

## To run JAX on TPU in Google Colab, uncomment the two lines below
# import jax.tools.colab_tpu
# jax.tools.colab_tpu.setup_tpu()

## JAX
import jax
import jax.numpy as jnp
from jax import random

## Flax (NN in JAX)
try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax
from flax import linen as nn
from flax.training import train_state, checkpoints

## Optax (Optimizers in JAX)
try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax

## PyTorch Data Loading
import torch
import torch.utils.data as data
import torchvision
from torchvision.datasets import CIFAR10

# Tensorboard extension (for visualization purposes later)
from torch.utils.tensorboard import SummaryWriter
%load_ext tensorboard

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../../saved_models/tutorial9_jax"
```

(continues on next page)

(continued from previous page)

```
print("Device:", jax.devices()[0])
Device: gpu:0
```

We have 4 pretrained models that we have to download. Remember to adjust the variables DATASET_PATH and CHECKPOINT_PATH if needed.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/JAX/tutorial9/"
# Files to download
pretrained_files = ["cifar10_64.ckpt", "cifar10_128.ckpt", "cifar10_256.ckpt", "cifar10_
˓→384.ckpt"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please contact the author with the full output,
˓→including the following error:\n", e)
```

In this tutorial, we work with the CIFAR10 dataset. In CIFAR10, each image has 3 color channels and is 32x32 pixels large. As autoencoders do not have the constraint of modeling images probabilistic, we can work on more complex image data (i.e. 3 color channels instead of black-and-white) much easier than for VAEs. In case you have downloaded CIFAR10 already in a different directory, make sure to set DATASET_PATH accordingly to prevent another download.

In contrast to previous tutorials on CIFAR10 like [Tutorial 5](#) (CNN classification), we do not normalize the data explicitly with a mean of 0 and std of 1, but roughly estimate it scaling the data between -1 and 1. This is because limiting the range will make our task of predicting/reconstructing images easier.

```
[3]: # Transformations applied on each image => bring them into a numpy array
def image_to_numpy(img):
    img = np.array(img, dtype=np.float32)
    if img.max() > 1:
        img = img / 255. * 2. - 1.
    return img

# For visualization, we might want to map JAX or numpy tensors back to PyTorch
def jax_to_torch(imgs):
    imgs = jax.device_get(imgs)
    imgs = torch.from_numpy(imgs.astype(np.float32))
    imgs = imgs.permute(0, 3, 1, 2)
    return imgs
```

(continues on next page)

(continued from previous page)

```

# We need to stack the batch elements
def numpy_collate(batch):
    if isinstance(batch[0], np.ndarray):
        return np.stack(batch)
    elif isinstance(batch[0], (tuple, list)):
        transposed = zip(*batch)
        return [numpy_collate(samples) for samples in transposed]
    else:
        return np.array(batch)

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=image_to_numpy,
                        download=True)
train_set, val_set = data.random_split(train_dataset, [45000, 5000], generator=torch.
                        Generator().manual_seed(42))

# Loading the test set
test_set = CIFAR10(root=DATASET_PATH, train=False, transform=image_to_numpy,
                    download=True)

# We define a set of data loaders that we can use for various purposes later.
train_loader = data.DataLoader(train_set, batch_size=256, shuffle=True, drop_last=True,
                               pin_memory=True, num_workers=4, collate_fn=numpy_collate, persistent_workers=True)
val_loader = data.DataLoader(val_set, batch_size=256, shuffle=False, drop_last=False,
                             num_workers=4, collate_fn=numpy_collate)
test_loader = data.DataLoader(test_set, batch_size=256, shuffle=False, drop_last=False,
                             num_workers=4, collate_fn=numpy_collate)

Files already downloaded and verified
Files already downloaded and verified

```

4.25.1 Building the autoencoder

In general, an autoencoder consists of an **encoder** that maps the input x to a lower-dimensional feature vector z , and a **decoder** that reconstructs the input \hat{x} from z . We train the model by comparing x to \hat{x} and optimizing the parameters to increase the similarity between x and \hat{x} . See below for a small illustration of the autoencoder framework.

We first start by implementing the encoder. The encoder effectively consists of a deep convolutional network, where we scale down the image layer-by-layer using strided convolutions. After downscaling the image three times, we flatten the features and apply linear layers. The latent representation z is therefore a vector of size d which can be flexibly selected.

```
[4]: class Encoder(nn.Module):
    c_hid : int
    latent_dim : int

    @nn.compact
    def __call__(self, x):
        x = nn.Conv(features=self.c_hid, kernel_size=(3, 3), strides=2)(x) # 32x32 =>
        # 16x16
```

(continues on next page)

(continued from previous page)

```

x = nn.gelu(x)
x = nn.Conv(features=self.c_hid, kernel_size=(3, 3))(x)
x = nn.gelu(x)
x = nn.Conv(features=2*self.c_hid, kernel_size=(3, 3), strides=2)(x) # 16x16 =>_
↪ 8x8
x = nn.gelu(x)
x = nn.Conv(features=2*self.c_hid, kernel_size=(3, 3))(x)
x = nn.gelu(x)
x = nn.Conv(features=2*self.c_hid, kernel_size=(3, 3), strides=2)(x) # 8x8 =>_
↪ 4x4
x = nn.gelu(x)
x = x.reshape(x.shape[0], -1) # Image grid to single feature vector
x = nn.Dense(features=self.latent_dim)(x)
return x

```

```
[5]: ## Test encoder implementation
# Random key for initialization
rng = random.PRNGKey(0)
# Example images as input
imgs = next(iter(train_loader))[0]
# Create encoder
encoder = Encoder(c_hid=32, latent_dim=128)
# Initialize parameters of encoder with random key and images
params = encoder.init(rng, imgs)['params']
# Apply encoder with parameters on the images
out = encoder.apply({'params': params}, imgs)
out.shape

del out, encoder, params
```

Note that we do not apply Batch Normalization here. This is because we want the encoding of each image to be independent of all the other images. Otherwise, we might introduce correlations into the encoding or decoding that we do not want to have. In some implementations, you still can see Batch Normalization being used, because it can also serve as a form of regularization. Nevertheless, the better practice is to go with other normalization techniques if necessary like Instance Normalization or Layer Normalization. Given the small size of the model, we can neglect normalization for now.

The decoder is a mirrored, flipped version of the encoder. The only difference is that we replace strided convolutions by transposed convolutions (i.e. deconvolutions) to upscale the features. Transposed convolutions can be imagined as adding the stride to the input instead of the output, and can thus upscale the input. For an illustration of a `nn.ConvTranspose` layer with kernel size 3, stride 2, and padding 1, see below (figure credit - [Vincent Dumoulin and Francesco Visin](#)):

You see that for an input of size 3×3 , we obtain an output of 5×5 . However, to truly have a reverse operation of the convolution, we need to ensure that the layer scales the input shape by a factor of 2 (e.g. $4 \times 4 \rightarrow 8 \times 8$). Flax already has this as a default setting for the padding, so we do not need to adjust anything here.

Overall, the decoder can be implemented as follows:

```
[6]: class Decoder(nn.Module):
    c_out : int
    c_hid : int
```

(continues on next page)

(continued from previous page)

```

latent_dim : int

@nn.compact
def __call__(self, x):
    x = nn.Dense(features=2*16*self.c_hid)(x)
    x = nn.gelu(x)
    x = x.reshape(x.shape[0], 4, 4, -1)
    x = nn.ConvTranspose(features=2*self.c_hid, kernel_size=(3, 3), strides=(2, 2))(x)
    x = nn.gelu(x)
    x = nn.Conv(features=2*self.c_hid, kernel_size=(3, 3))(x)
    x = nn.gelu(x)
    x = nn.ConvTranspose(features=self.c_hid, kernel_size=(3, 3), strides=(2, 2))(x)
    x = nn.gelu(x)
    x = nn.Conv(features=self.c_hid, kernel_size=(3, 3))(x)
    x = nn.gelu(x)
    x = nn.ConvTranspose(features=self.c_out, kernel_size=(3, 3), strides=(2, 2))(x)
    x = nn.tanh(x)
    return x

```

```

[7]: ## Test decoder implementation
# Random key for initialization
rng = random.PRNGKey(0)
# Example latents as input
rng, lat_rng = random.split(rng)
latents = random.normal(lat_rng, (16, 128))
# Create decoder
decoder = Decoder(c_hid=32, latent_dim=128, c_out=3)
# Initialize parameters of decoder with random key and latents
rng, init_rng = random.split(rng)
params = decoder.init(init_rng, latents)['params']
# Apply decoder with parameters on the images
out = decoder.apply({'params': params}, latents)
out.shape

del out, decoder, params

```

The encoder and decoder networks we chose here are relatively simple. Usually, more complex networks are applied, especially when using a ResNet-based architecture. For example, see [VQ-VAE](#) and [NVAE](#) (although the papers discuss architectures for VAEs, they can equally be applied to standard autoencoders).

In a final step, we add the encoder and decoder together into the autoencoder architecture.

```

[8]: class Autoencoder(nn.Module):
    c_hid: int
    latent_dim : int

    def setup(self):
        # Alternative to @nn.compact -> explicitly define modules
        # Better for later when we want to access the encoder and decoder explicitly
        self.encoder = Encoder(c_hid=self.c_hid, latent_dim=self.latent_dim)
        self.decoder = Decoder(c_hid=self.c_hid, latent_dim=self.latent_dim, c_out=3)

```

(continues on next page)

(continued from previous page)

```
def __call__(self, x):
    z = self.encoder(x)
    x_hat = self.decoder(z)
    return x_hat
```

```
[9]: ## Test Autoencoder implementation
# Random key for initialization
rng = random.PRNGKey(0)
# Example images as input
imgs = next(iter(train_loader))[0]
# Create encoder
autoencoder = Autoencoder(c_hid=32, latent_dim=128)
# Initialize parameters of encoder with random key and images
params = autoencoder.init(rng, imgs)['params']
# Apply encoder with parameters on the images
out = autoencoder.apply({'params': params}, imgs)
out.shape

del out, autoencoder, params
```

For the loss function, we use the mean squared error (MSE), which we implement below:

```
[10]: def mse_recon_loss(model, params, batch):
    imgs, _ = batch
    recon_imgs = model.apply({'params': params}, imgs)
    loss = ((recon_imgs - imgs) ** 2).mean(axis=0).sum() # Mean over batch, sum over pixels
    return loss
```

The mean squared error pushes the network to pay special attention to those pixel values its estimate is far away. Predicting 127 instead of 128 is not important when reconstructing, but confusing 0 with 128 is much worse. Note that in contrast to VAEs, we do not predict the probability per pixel value, but instead use a distance measure. This saves a lot of parameters and simplifies training. To get a better intuition per pixel, we report the summed squared error averaged over the batch dimension (any other mean/sum leads to the same result/parameters).

However, MSE has also some considerable disadvantages. Usually, MSE leads to blurry images where small noise/high-frequent patterns are removed as those cause a very low error. To ensure realistic images to be reconstructed, one could combine Generative Adversarial Networks (lecture 10) with autoencoders as done in several works (e.g. see [here](#), [here](#) or these [slides](#)). Additionally, comparing two images using MSE does not necessarily reflect their visual similarity. For instance, suppose the autoencoder reconstructs an image shifted by one pixel to the right and bottom. Although the images are almost identical, we can get a higher loss than predicting a constant pixel value for half of the image (see code below). An example solution for this issue includes using a separate, pre-trained CNN, and use a distance of visual features in lower layers as a distance measure instead of the original pixel-level comparison.

```
[11]: def compare_imgs(img1, img2, title_prefix=''):
    # Calculate MSE loss between both images
    loss = ((img1 - img2) ** 2).sum()
    # Plot images for visual comparison
    imgs = jax_to_torch(np.stack([img1, img2], axis=0))
    grid = torchvision.utils.make_grid(imgs, nrow=2, normalize=True, value_range=(-1,1))
    grid = grid.permute(1, 2, 0)
    plt.figure(figsize=(4,2))
```

(continues on next page)

(continued from previous page)

```
plt.title(f"{title_prefix} Loss: {loss.item():.2f}")
plt.imshow(grid)
plt.axis('off')
plt.show()

for i in range(2):
    # Load example image
    img, _ = train_dataset[i]
    img_mean = img.mean(axis=(0,1), keepdims=True)

    # Shift image by one pixel
    SHIFT = 1
    img_shifted = np.roll(img, shift=SHIFT, axis=0)
    img_shifted = np.roll(img_shifted, shift=SHIFT, axis=1)
    img_shifted[:1,:,:] = img_mean
    img_shifted[:,1:,:] = img_mean
    compare_imgs(img, img_shifted, "Shifted -")

    # Set half of the image to zero
    img_masked = np.copy(img)
    img_masked[:img_masked.shape[1]//2,:,:] = img_mean
    compare_imgs(img, img_masked, "Masked -")
```

Shifted - Loss: 205.40



Masked - Loss: 158.48



Shifted - Loss: 418.47



Masked - Loss: 295.20



Training the model

During the training, we want to keep track of the learning progress by seeing reconstructions made by our model. For this, we implement a callback object which will add reconstructions every N epochs to our tensorboard. To align it with the PyTorch tutorial version, we implement it similar to how we would do it in PyTorch Lightning:

```
[12]: class GenerateCallback:

    def __init__(self, input_imgs, every_n_epochs=1):
        super().__init__()
        self.input_imgs = input_imgs # Images to reconstruct during training
        self.every_n_epochs = every_n_epochs # Only save those images every N epochs,
        ↴(otherwise tensorboard gets quite large)

    def log_generations(self, model, state, logger, epoch):
        if epoch % self.every_n_epochs == 0:
            reconst_imgs = model.apply({'params': state.params}, self.input_imgs)
            reconst_imgs = jax.device_get(reconst_imgs)

            # Plot and add to tensorboard
            imgs = np.stack([self.input_imgs, reconst_imgs], axis=1).reshape(-1, *self.
            ↴input_imgs.shape[1:])
            imgs = jax_to_torch(imgs)
            grid = torchvision.utils.make_grid(imgs, nrow=2, normalize=True, value_
            ↴range=(-1, 1))
            logger.add_image("Reconstructions", grid, global_step=epoch)
```

Further, to train multiple models with different hyperparameters, we summarize all training functionalities in a trainer

object below:

```
[13]: class TrainerModule:

    def __init__(self, c_hid, latent_dim, lr=1e-3, seed=42):
        super().__init__()
        self.c_hid = c_hid
        self.latent_dim = latent_dim
        self.lr = lr
        self.seed = seed
        # Create empty model. Note: no parameters yet
        self.model = Autoencoder(c_hid=self.c_hid, latent_dim=self.latent_dim)
        # Prepare logging
        self.exmp_imgs = next(iter(val_loader))[0][:8]
        self.log_dir = os.path.join(CHECKPOINT_PATH, f'cifar10_{self.latent_dim}')
        self.generate_callback = GenerateCallback(self.exmp_imgs, every_n_epochs=50)
        self.logger = SummaryWriter(log_dir=self.log_dir)
        # Create jitted training and eval functions
        self.create_functions()
        # Initialize model
        self.init_model()

    def create_functions(self):
        # Training function
        def train_step(state, batch):
            loss_fn = lambda params: mse_recon_loss(self.model, params, batch)
            loss, grads = jax.value_and_grad(loss_fn)(state.params) # Get loss and
            gradients for loss
            state = state.apply_gradients(grads=grads) # Optimizer update step
            return state, loss
        self.train_step = jax.jit(train_step)
        # Eval function
        def eval_step(state, batch):
            return mse_recon_loss(self.model, state.params, batch)
        self.eval_step = jax.jit(eval_step)

    def init_model(self):
        # Initialize model
        rng = jax.random.PRNGKey(self.seed)
        rng, init_rng = jax.random.split(rng)
        params = self.model.init(init_rng, self.exmp_imgs)['params']
        # Initialize learning rate schedule and optimizer
        lr_schedule = optax.warmup_cosine_decay_schedule(
            init_value=0.0,
            peak_value=1e-3,
            warmup_steps=100,
            decay_steps=500*len(train_loader),
            end_value=1e-5
        )
        optimizer = optax.chain(
            optax.clip(1.0), # Clip gradients at 1
            optax.adam(lr_schedule)
        )
```

(continues on next page)

(continued from previous page)

```

# Initialize training state
self.state = train_state.TrainState.create(apply_fn=self.model.apply,
                                          params=params, tx=optimizer)

def train_model(self, num_epochs=500):
    # Train model for defined number of epochs
    best_eval = 1e6
    for epoch_idx in tqdm(range(1, num_epochs+1)):
        self.train_epoch(epoch=epoch_idx)
        if epoch_idx % 10 == 0:
            eval_loss = self.eval_model(val_loader)
            self.logger.add_scalar('val/loss', eval_loss, global_step=epoch_idx)
            if eval_loss < best_eval:
                best_eval = eval_loss
                self.save_model(step=epoch_idx)
            self.generate_callback.log_generations(self.model, self.state,
                                                    logger=self.logger, epoch=epoch_idx)
            self.logger.flush()

def train_epoch(self, epoch):
    # Train model for one epoch, and log avg loss
    losses = []
    for batch in train_loader:
        self.state, loss = self.train_step(self.state, batch)
        losses.append(loss)
    losses_np = np.stack(jax.device_get(losses))
    avg_loss = losses_np.mean()
    self.logger.add_scalar('train/loss', avg_loss, global_step=epoch)

def eval_model(self, data_loader):
    # Test model on all images of a data loader and return avg loss
    losses = []
    batch_sizes = []
    for batch in data_loader:
        loss = self.eval_step(self.state, batch)
        losses.append(loss)
        batch_sizes.append(batch[0].shape[0])
    losses_np = np.stack(jax.device_get(losses))
    batch_sizes_np = np.stack(batch_sizes)
    avg_loss = (losses_np * batch_sizes_np).sum() / batch_sizes_np.sum()
    return avg_loss

def save_model(self, step=0):
    # Save current model at certain training iteration
    checkpoints.save_checkpoint(ckpt_dir=self.log_dir, target=self.state.params,
                               prefix=f'cifar10_{self.latent_dim}_', step=step)

def load_model(self, pretrained=False):
    # Load model. We use different checkpoint for pretrained models
    if not pretrained:
        params = checkpoints.restore_checkpoint(ckpt_dir=self.log_dir, target=self.
                                                state.params, prefix=f'cifar10_{self.latent_dim}_')

```

(continues on next page)

(continued from previous page)

```

else:
    params = checkpoints.restore_checkpoint(ckpt_dir=os.path.join(CHECKPOINT_PATH,
                                                               f'cifar10_{self.latent_dim}.ckpt'), target=self.state.params)
    self.state = train_state.TrainState.create(apply_fn=self.model.apply,
                                                params=params, tx=self.state.tx)

def checkpoint_exists(self):
    # Check whether a pretrained model exist for this autoencoder
    return os.path.isfile(os.path.join(CHECKPOINT_PATH, f'cifar10_{self.latent_dim}.ckpt'))

```

We will now write a training function that allows us to train the autoencoder with different latent dimensionality and returns the test score. We provide pre-trained models and recommend you using those, especially when you work on a computer without GPU. Of course, feel free to train your own models.

```
[14]: def train_cifar(latent_dim):
    # Create a trainer module with specified hyperparameters
    trainer = TrainerModule(c_hid=32, latent_dim=latent_dim)
    if not trainer.checkpoint_exists(): # Skip training if pretrained model exists
        trainer.train_model(num_epochs=500)
        trainer.load_model()
    else:
        trainer.load_model(pretrained=True)
    test_loss = trainer.eval_model(test_loader)
    # Bind parameters to model for easier inference
    trainer.model_bd = trainer.model.bind({'params': trainer.state.params})
    return trainer, test_loss
```

Comparing latent dimensionality

When training an autoencoder, we need to choose a dimensionality for the latent representation z . The higher the latent dimensionality, the better we expect the reconstruction to be. However, the idea of autoencoders is to *compress* data. Hence, we are also interested in keeping the dimensionality low. To find the best tradeoff, we can train multiple models with different latent dimensionalities. The original input has $32 \times 32 \times 3 = 3072$ pixels. Keeping this in mind, a reasonable choice for the latent dimensionality might be between 64 and 384:

```
[15]: model_dict = {}
for latent_dim in [64, 128, 256, 384]:
    trainer_ld, test_loss_ld = train_cifar(latent_dim)
    model_dict[latent_dim] = {"trainer": trainer_ld, "result": test_loss_ld}
```

After training the models, we can plot the reconstruction loss over the latent dimensionality to get an intuition how these two properties are correlated:

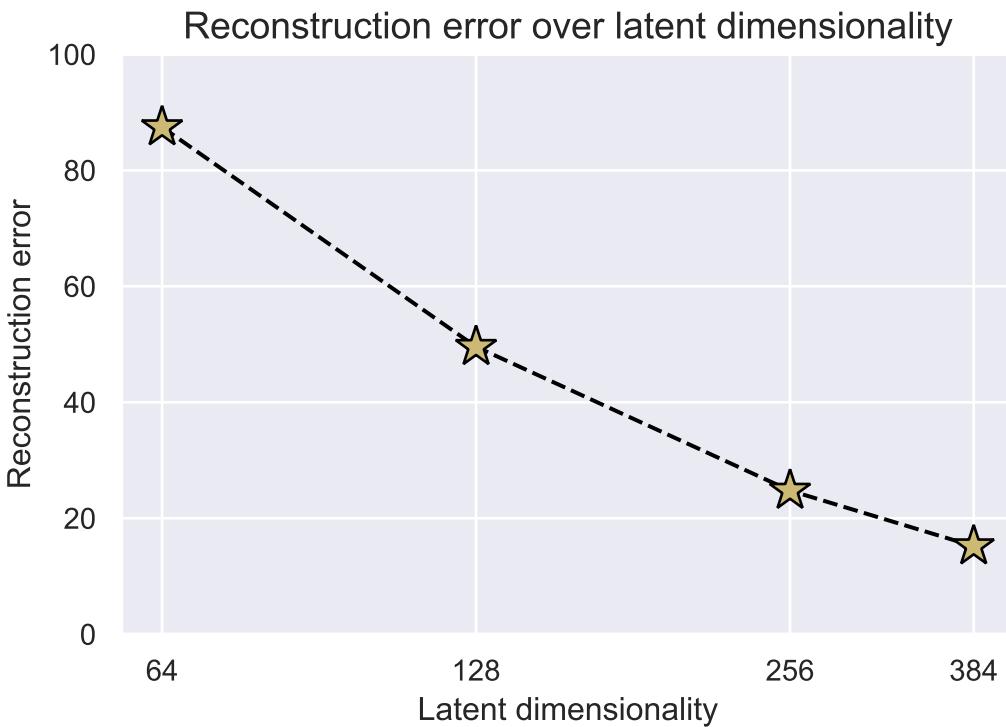
```
[16]: latent_dims = sorted([k for k in model_dict])
val_scores = [model_dict[k]["result"] for k in latent_dims]

fig = plt.figure(figsize=(6,4))
plt.plot(latent_dims, val_scores, '--', color="#000", marker="*", markeredgecolor="#000",
          markerfacecolor="y", markersize=16)
plt.xscale("log")
```

(continues on next page)

(continued from previous page)

```
plt.xticks(latent_dims, labels=latent_dims)
plt.title("Reconstruction error over latent dimensionality", fontsize=14)
plt.xlabel("Latent dimensionality")
plt.ylabel("Reconstruction error")
plt.minorticks_off()
plt.ylim(0,100)
plt.show()
```



As we initially expected, the reconstruction loss goes down with increasing latent dimensionality. For our model and setup, the two properties seem to be exponentially (or double exponentially) correlated. To understand what these differences in reconstruction error mean, we can visualize example reconstructions of the four models. For simplicity, we visualize four training images of CIFAR10 we have seen already before. For larger models that may overfit, it is recommended to use images from the validation set.

```
[17]: def visualize_reconstructions(trainer, input_imgs):
    # Reconstruct images
    reconst_imgs = trainer.model_bd(input_imgs)
    imgs = np.stack([input_imgs, reconst_imgs], axis=1).reshape(-1, *reconst_imgs.
    ↵shape[1:])
    # Plotting
    imgs = jax_to_torch(imgs)
    grid = torchvision.utils.make_grid(imgs, nrow=4, normalize=True, value_range=(-1,1))
    grid = grid.permute(1, 2, 0)
    plt.figure(figsize=(7,4.5))
    plt.title(f'Reconstructed from {trainer.latent_dim} latents')
    plt.imshow(grid)
    plt.axis('off')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

```
[18]: input_imgs = np.stack([image_to_numpy(train_dataset[i][0]) for i in range(4)], axis=0)
for latent_dim in model_dict:
    visualize_reconstructions(model_dict[latent_dim]["trainer"], input_imgs)
```

Reconstructed from 64 latents



Reconstructed from 128 latents



Reconstructed from 256 latents



Reconstructed from 384 latents



Clearly, the smallest latent dimensionality can only save information about the rough shape and color of the object, but the reconstructed image is extremely blurry and it is hard to recognize the original object in the reconstruction. With 128 features, we can recognize some shapes again although the picture remains blurry. The models with the highest two dimensionalities reconstruct the images quite well. The difference between 256 and 384 is marginal at first sight but can be noticed when comparing, for instance, the backgrounds of the first image (the 384 features model more of the pattern than 256).

Out-of-distribution images

Before continuing with the applications of autoencoder, we can actually explore some limitations of our autoencoder. For example, what happens if we try to reconstruct an image that is clearly out of the distribution of our dataset? We expect the decoder to have learned some common patterns in the dataset, and thus might in particular fail to reconstruct images that do not follow these patterns.

The first experiment we can try is to reconstruct noise. We, therefore, create two images whose pixels are randomly sampled from a uniform distribution over pixel values, and visualize the reconstruction of the model (feel free to test different latent dimensionalities):

```
[19]: rng = jax.random.PRNGKey(123)
rgn, noise_rgn = jax.random.split(rng)
rand_imgs = jax.random.uniform(rng, (2, 32, 32, 3)) * 2 - 1
visualize_reconstructions(model_dict[256]["trainer"], rand_imgs)
```

Reconstructed from 256 latents



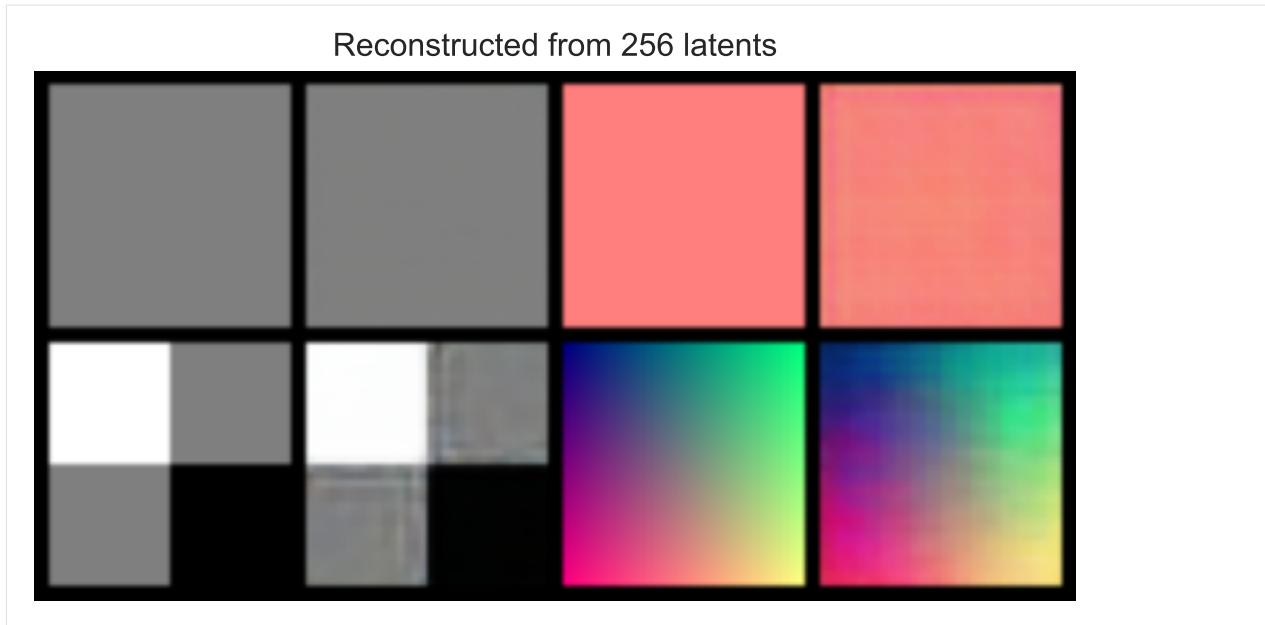
The reconstruction of the noise is quite poor, and seems to introduce some rough patterns. As the input does not follow the patterns of the CIFAR dataset, the model has issues reconstructing it accurately.

We can also check how well the model can reconstruct other manually-coded patterns:

```
[20]: # JAX arrays are natively immutable. This is why we first create the images in numpy, ↵
       and push them to JAX afterwards
plain_imgs = np.zeros((4, 32, 32, 3))

# Single color channel
plain_imgs[1,:,:,:0] = 1
# Checkboard pattern
plain_imgs[2,:16,:16] = 1
plain_imgs[2,16:,:16:] = -1
# Color progression
xx, yy = np.meshgrid(np.linspace(-1,1,32), np.linspace(-1,1,32), indexing='ij')
plain_imgs[3,:,:,:0] = xx
plain_imgs[3,:,:,:1] = yy

visualize_reconstructions(model_dict[256]["trainer"], plain_imgs)
```



The plain, constant images are reconstructed relatively good although the single color channel contains some noticeable noise. The hard borders of the checkboard pattern are not as sharp as intended, as well as the color progression, both because such patterns never occur in the real-world pictures of CIFAR.

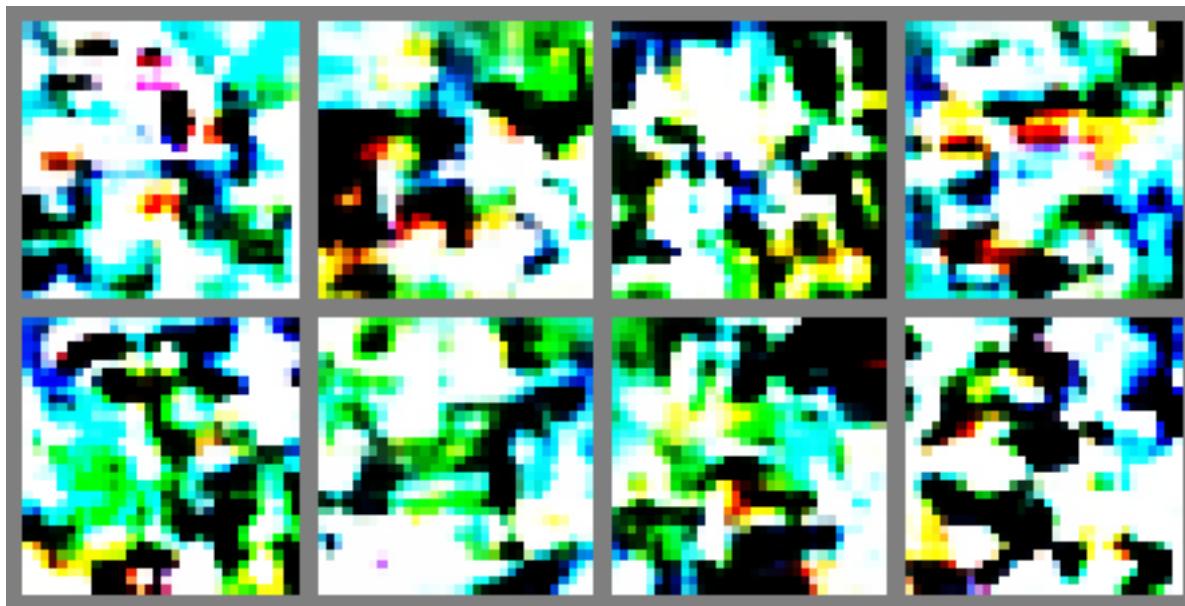
In general, autoencoders tend to fail reconstructing high-frequent noise (i.e. sudden, big changes across few pixels) due to the choice of MSE as loss function (see our previous discussion about loss functions in autoencoders). Small misalignments in the decoder can lead to huge losses so that the model settles for the expected value/mean in these regions. For low-frequent noise, a misalignment of a few pixels does not result in a big difference to the original image. However, the larger the latent dimensionality becomes, the more of this high-frequent noise can be accurately reconstructed.

Generating new images

Variational autoencoders are a generative version of the autoencoders because we regularize the latent space to follow a Gaussian distribution. However, in vanilla autoencoders, we do not have any restrictions on the latent vector. So what happens if we would actually input a randomly sampled latent vector into the decoder? Let's find it out below:

```
[21]: trainer = model_dict[256]["trainer"]
rng, latent_noise = jax.random.split(rng)
latent_vectors = jax.random.normal(latent_noise, (8, trainer.latent_dim))
# Decode images -> Run model.decode method of the trainer's model with given parameters
imgs = trainer.model_bd.decoder(latent_vectors) # nn.apply(lambda model: model.
                                                # decode(latent_vectors), trainer.model)({'params': trainer.state.params})

imgs = jax_to_torch(imgs)
grid = torchvision.utils.make_grid(imgs, nrow=4, normalize=True, value_range=(-1,1), pad_
                                   value=0.5)
grid = grid.permute(1, 2, 0)
plt.figure(figsize=(8,5))
plt.imshow(grid)
plt.axis('off')
plt.show()
```



As we can see, the generated images more look like art than realistic images. As the autoencoder was allowed to structure the latent space in whichever way it suits the reconstruction best, there is no incentive to map every possible latent vector to realistic images. Furthermore, the distribution in latent space is unknown to us and doesn't necessarily follow a multivariate normal distribution. Thus, we can conclude that vanilla autoencoders are indeed not generative.

4.25.2 Finding visually similar images

One application of autoencoders is to build an image-based search engine to retrieve visually similar images. This can be done by representing all images as their latent dimensionality, and find the closest K images in this domain. The first step to such a search engine is to encode all images into z . In the following, we will use the training set as a search corpus, and the test set as queries to the system.

Warning: the following cells can be computationally heavy for a weak CPU-only system. If you do not have a strong computer and are not on Google Colab, you might want to skip the execution of the following cells and rely on the results shown in the filled notebook.

```
[22]: # We use the following model throughout this section.  
# If you want to try a different latent dimensionality, change it here!  
trainer = model_dict[128]["trainer"]
```

```
[23]: def embed_imgs(trainer, data_loader):  
    # Encode all images in the data_laoder using model, and return both images and  
    # encodings  
    img_list, embed_list = [], []  
  
    @jax.jit  
    def encode(imgs):  
        return trainer.model_bd.encoder(imgs)
```

(continues on next page)

(continued from previous page)

```

for imgs, _ in tqdm(data_loader, desc="Encoding images", leave=False):
    z = encode(imgs)
    z = jax.device_get(z)
    imgs = jax.device_get(imgs)
    img_list.append(imgs)
    embed_list.append(z)
return (np.concatenate(img_list, axis=0), np.concatenate(embed_list, axis=0))

train_img_embeds = embed_imgs(trainer, train_loader)
test_img_embeds = embed_imgs(trainer, test_loader)

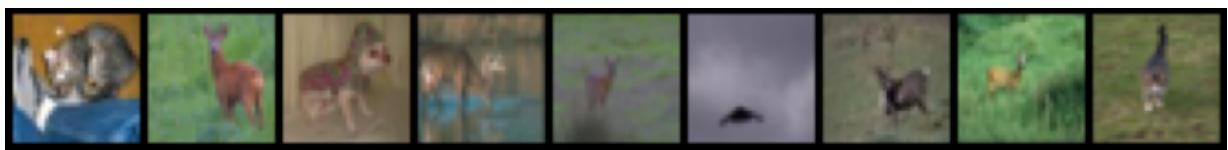
Encoding images:  0% | 0/175 [00:00<?, ?it/s]
Encoding images:  0% | 0/40 [00:00<?, ?it/s]

```

After encoding all images, we just need to write a function that finds the closest K images and returns (or plots) those:

```
[24]: def find_similar_images(query_img, query_z, key_embeds, K=8):
    # Find closest K images. We use the euclidean distance here but other like cosine_
    # distance can also be used.
    dist = np.linalg.norm(query_z[None,:] - key_embeds[1], axis=-1)
    indices = np.argsort(dist)
    dist = dist[indices]
    # Plot K closest images
    imgs_to_display = np.concatenate([query_img[None], key_embeds[0][indices[:K]]],_
    axis=0)
    imgs_to_display = torch.from_numpy(imgs_to_display)
    imgs_to_display = imgs_to_display.permute(0, 3, 1, 2)
    grid = torchvision.utils.make_grid(imgs_to_display, nrow=K+1, normalize=True, value_
    range=(-1,1))
    grid = grid.permute(1, 2, 0)
    plt.figure(figsize=(12,3))
    plt.imshow(grid)
    plt.axis('off')
    plt.show()
```

```
[25]: # Plot the closest images for the first N test images as example
for i in range(8):
    find_similar_images(test_img_embeds[0][i], test_img_embeds[1][i], key_embeds=train_
    img_embeds)
```





Based on our autoencoder, we see that we are able to retrieve many similar images to the test input. In particular, in row 4, we can spot that some test images might not be that different from the training set as we thought (same poster, just different scaling/color scaling). We also see that although we haven't given the model any labels, it can cluster different classes in different parts of the latent space (airplane + ship, animals, etc.). This is why autoencoders can also be used as a pre-training strategy for deep networks, especially when we have a large set of unlabeled images (often the case). However, it should be noted that the background still plays a big role in autoencoders while it doesn't for classification. Hence, we don't get "perfect" clusters and need to finetune such models for classification.

Tensorboard clustering

Another way of exploring the similarity of images in the latent space is by dimensionality-reduction methods like PCA or T-SNE. Luckily, Tensorboard provides a nice interface for this and we can make use of it in the following:

```
[26]: # We use the following model throughout this section.
# If you want to try a different latent dimensionality, change it here!
trainer = model_dict[128]["trainer"]
```

```
[27]: # Create a summary writer
writer = SummaryWriter("tensorboard/")
```

The function `add_embedding` allows us to add high-dimensional feature vectors to TensorBoard on which we can perform clustering. What we have to provide in the function are the feature vectors, additional metadata such as the labels, and the original images so that we can identify a specific image in the clustering.

```
[28]: ## In case you obtain the following error in the next cell, execute the import_
statements and last line in this cell
## AttributeError: module 'tensorflow._api.v2.io.gfile' has no attribute 'get_filesystem'

import tensorflow as tf
import tensorboard as tb
tf.io.gfile = tb.compat.tensorflow_stub.io.gfile
```

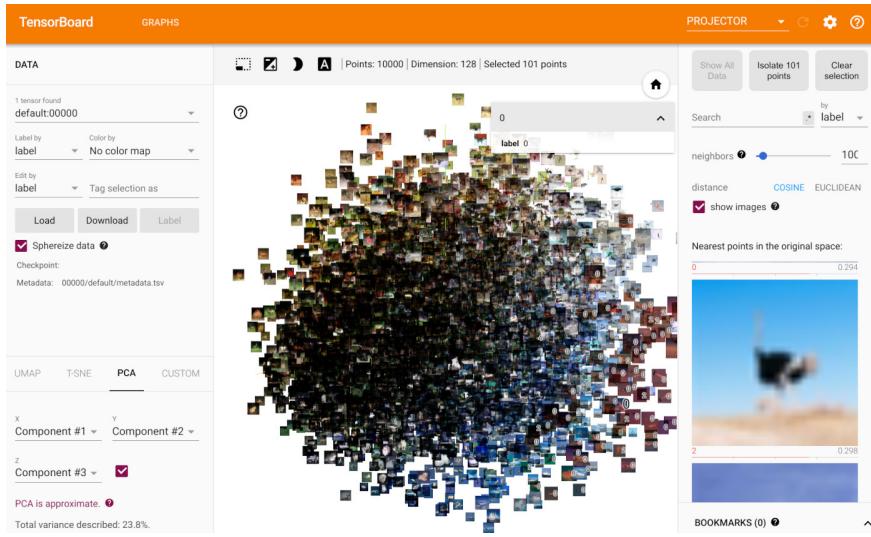
```
[29]: # Note: the embedding projector in tensorboard is computationally heavy.
# Reduce the image amount below if your computer struggles with visualizing all 10k_
points
NUM_IMGS = len(test_set)

writer.add_embedding(test_img_embeds[1][:NUM_IMGS], # Encodings per image
                     metadata=[test_set[i][1] for i in range(NUM_IMGS)], # Adding the_
                     labels per image to the plot
                     label_img=torch.from_numpy(test_img_embeds[0][:NUM_IMGS]+1).
                     permute(0, 3, 1, 2)/2.0) # Adding the original images to the plot
```

Finally, we can run tensorboard to explore similarities among images:

```
[30]: %tensorboard --logdir tensorboard/
```

You should be able to see something similar as in the following image. In case the projector stays empty, try to start the TensorBoard outside of the Jupyter notebook.



Overall, we can see that the model indeed clustered images together that are visually similar. Especially the background color seems to be a crucial factor in the encoding. This correlates to the chosen loss function, here Mean Squared Error on pixel-level because the background is responsible for more than half of the pixels in an average image. Hence, the model learns to focus on it. Nevertheless, we can see that the encodings also separate a couple of classes in the latent space although it hasn't seen any labels. This shows again that autoencoding can also be used as a “pre-training”/transfer learning task before classification.

```
[31]: # Closing the summary writer
writer.close()
```

4.25.3 Conclusion

In this tutorial, we have implemented our own autoencoder on small RGB images and explored various properties of the model. In contrast to variational autoencoders, vanilla AEs are not generative and can work on MSE loss functions. This makes them often easier to train. Both versions of AE can be used for dimensionality reduction, as we have seen for finding visually similar images beyond pixel distances. Despite autoencoders gaining less interest in the research community due to their more “theoretically” challenging counterpart of VAEs, autoencoders still find usage in a lot of applications like denoising and compression. Hence, AEs are an essential tool that every Deep Learning engineer/researcher should be familiar with.

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.26 Tutorial 11 (JAX): Normalizing Flows for image modeling

Filled notebook:

Pre-trained models:

PyTorch version:

Author: Phillip Lippe

Note: This notebook is written in JAX+Flax. It is a 1-to-1 translation of the original notebook written in PyTorch+PyTorch Lightning with almost identical results. For an introduction to JAX, check out our [Tutorial 2 \(JAX\): Introduction to JAX+Flax](#). Further, throughout the notebook, we comment on major differences to the PyTorch version and provide explanations for the major parts of the JAX code.

Speed comparison: We note the training times for all models in the PyTorch and the JAX implementation below (PyTorch v1.11, JAX v0.3.13). The models were trained on the same hardware (NVIDIA RTX3090, 24 core CPU) and we slightly adjusted the tutorials to use the exact same training settings (200 epochs, data loading parameters, evaluation schedule, etc.). Overall, the JAX implementation is *2.0-2.5x faster* than PyTorch! Different network architectures may have different speedups.

Models	PyTorch	JAX
MNIST Flow - Simple	2hrs 37min 29sec	1hrs 17min 59sec
MNIST Flow - VarDeq	3hrs 25min 10sec	1hrs 36min 56sec
MNIST Flow - Multiscale	2hrs 17min 10sec	57min 57sec

In this tutorial, we will take a closer look at complex, deep normalizing flows. The most popular, current application of deep normalizing flows is to model datasets of images. As for other generative models, images are a good domain to start working on because (1) CNNs are widely studied and strong models exist, (2) images are high-dimensional and complex, and (3) images are discrete integers. In this tutorial, we will review current advances in normalizing flows for image modeling, and get hands-on experience on coding normalizing flows. Note that normalizing flows are commonly parameter heavy and therefore computationally expensive. We will use relatively simple and shallow flows to save computational cost and allow you to run the notebook on CPU, but keep in mind that a simple way to improve the scores of the flows we study here is to make them deeper. The first cell imports our usual libraries.

```
[1]: ## Standard libraries
import os
import math
import time
import json
import numpy as np
from typing import Sequence

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
```

(continues on next page)

(continued from previous page)

```
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## Progress bar
from tqdm.notebook import tqdm

## To run JAX on TPU in Google Colab, uncomment the two lines below
# import jax.tools.colab_tpu
# jax.tools.colab_tpu.setup_tpu()

## JAX
import jax
import jax.numpy as jnp
from jax import random

## Flax (NN in JAX)
try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax
from flax import linen as nn
from flax.training import train_state, checkpoints

## Optax (Optimizers in JAX)
try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax

## PyTorch Data Loading
import torch
import torch.utils.data as data
from torch.utils.tensorboard import SummaryWriter
import torchvision
from torchvision.datasets import MNIST

# Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../../saved_models/tutorial11_jax"

# Seeding for random operations
main_rng = random.PRNGKey(42)

print("Device:", jax.devices()[0])
```

```
/home/phillip/anaconda3/envs/dl2020/lib/python3.7/site-packages/chex/_src/pytypes.py:37: FutureWarning: jax.tree_structure is deprecated, and will be removed in a future release. Use jax.tree_util.tree_structure instead.
PyTreeDef = type(jax.tree_structure(None))
WARNING:absl:GlobalAsyncCheckpointManager is not imported correctly. Checkpointing of GlobalDeviceArrays will not be available. To use the feature, install tensorstore.

Device: gpu:0
```

Again, we have a few pretrained models. We download them below to the specified path above.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/JAX/tutorial11/"
# Files to download
pretrained_files = ["MNISTFlow_simple.ckpt", "MNISTFlow_vardeq.ckpt", "MNISTFlow_multiscale.ckpt",
                     "MNISTFlow_simple_results.json", "MNISTFlow_vardeq_results.json",
                     "MNISTFlow_multiscale_results.json"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please contact the author with the full output, including the following error:\n", e)
```

We will use the MNIST dataset in this notebook. MNIST constitutes, despite its simplicity, a challenge for small generative models as it requires the global understanding of an image. At the same time, we can easily judge whether generated images come from the same distribution as the dataset (i.e. represent real digits), or not.

To deal better with the discrete nature of the images, we transform them from a range of 0-1 to a range of 0-255 as integers.

```
[3]: # Transformations applied on each image => bring them into a numpy array
# Note that we keep them in the range 0-255 (integers)
def image_to_numpy(img):
    img = np.array(img, dtype=np.int32)
    img = img[..., None] # Make image [28, 28, 1]
    return img

# We need to stack the batch elements
def numpy_collate(batch):
    if isinstance(batch[0], np.ndarray):
        return np.stack(batch)
    elif isinstance(batch[0], (tuple, list)):
```

(continues on next page)

(continued from previous page)

```

transposed = zip(*batch)
return [numpy_collate(samples) for samples in transposed]
else:
    return np.array(batch)

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = MNIST(root=DATASET_PATH, train=True, transform=image_to_numpy,
download=True)
train_set, val_set = torch.utils.data.random_split(train_dataset, [50000, 10000],
generator=torch.Generator().manual_
seed(42))

# Loading the test set
test_set = MNIST(root=DATASET_PATH, train=False, transform=image_to_numpy, download=True)

# We define a set of data loaders that we can use for various purposes
# Data loader for loading examples throughout the notebook
train_exmp_loader = data.DataLoader(train_set, batch_size=256, shuffle=False, drop_
last=False, collate_fn=numpy_collate)
# Actual data loaders for training, validation, and testing
train_data_loader = data.DataLoader(train_set,
batch_size=128,
shuffle=True,
drop_last=True,
collate_fn=numpy_collate,
num_workers=8,
persistent_workers=True)
val_loader = data.DataLoader(val_set, batch_size=64, shuffle=False, drop_last=False, num_
workers=4, collate_fn=numpy_collate)
test_loader = data.DataLoader(test_set, batch_size=64, shuffle=False, drop_last=False,_
num_workers=4, collate_fn=numpy_collate)

```

In addition, we will define below a function to simplify the visualization of images/samples. Some training examples of the MNIST dataset is shown below.

```
[4]: def show_imgs(imgs, title=None, row_size=4):
    # Form a grid of pictures (we use max. 8 columns)
    imgs = np.copy(jax.device_get(imgs))
    num_imgs = imgs.shape[0]
    is_int = (imgs.dtype==np.int32)
    nrow = min(num_imgs, row_size)
    ncol = int(math.ceil(num_imgs/nrow))
    imgs_torch = torch.from_numpy(imgs).permute(0, 3, 1, 2)
    imgs = torchvision.utils.make_grid(imgs_torch, nrow=nrow, pad_value=128 if is_int_
else 0.5)
    np_imgs = imgs.cpu().numpy()
    # Plot the grid
    plt.figure(figsize=(1.5*nrow, 1.5*ncol))
    plt.imshow(np.transpose(np_imgs, (1, 2, 0)), interpolation='nearest')
    plt.axis('off')
    if title is not None:
        plt.title(title)
```

(continues on next page)

(continued from previous page)

```

plt.show()
plt.close()

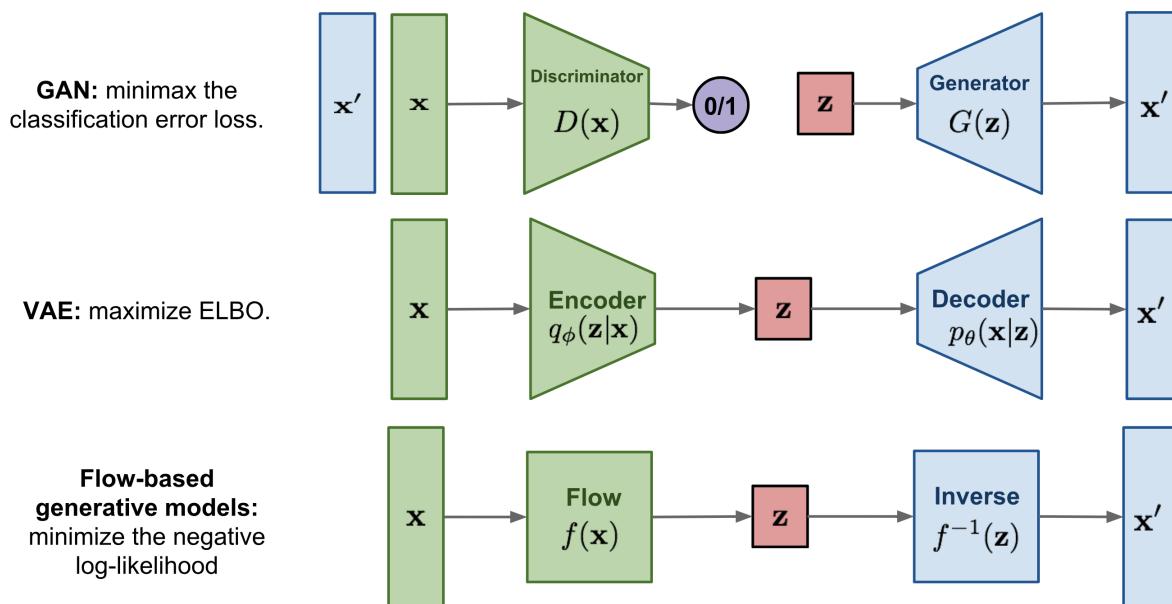
show_imgs(np.stack([train_set[i][0] for i in range(8)], axis=0))

```



4.26.1 Normalizing Flows as generative model

In the previous lectures, we have seen Energy-based models, Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs) as example of generative models. However, none of them explicitly learn the probability density function $p(x)$ of the real input data. While VAEs model a lower bound, energy-based models only implicitly learn the probability density. GANs on the other hand provide us a sampling mechanism for generating new data, without offering a likelihood estimate. The generative model we will look at here, called Normalizing Flows, actually models the true data distribution $p(x)$ and provides us with an exact likelihood estimate. Below, we can visually compare VAEs, GANs and Flows (figure credit - [Lilian Weng](#)):



The major difference compared to VAEs is that flows use *invertible* functions f to map the input data x to a latent

representation z . To realize this, z must be of the same shape as x . This is in contrast to VAEs where z is usually much lower dimensional than the original input data. However, an invertible mapping also means that for every data point x , we have a corresponding latent representation z which allows us to perform lossless reconstruction (z to x). In the visualization above, this means that $x = x'$ for flows, no matter what invertible function f and input x we choose.

Nonetheless, how are normalizing flows modeling a probability density with an invertible function? The answer to this question is the rule for change of variables. Specifically, given a prior density $p_z(z)$ (e.g. Gaussian) and an invertible function f , we can determine $p_x(x)$ as follows:

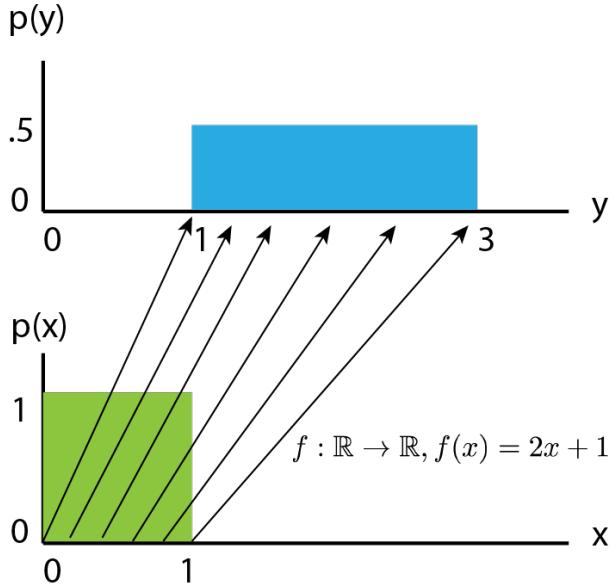
$$\int p_x(x)dx = \int p_z(z)dz = 1 \quad (\text{by definition of a probability distribution})$$

$$\Leftrightarrow p_x(x) = p_z(z) \left| \frac{dz}{dx} \right| = p_z(f(x)) \left| \frac{df(x)}{dx} \right|$$

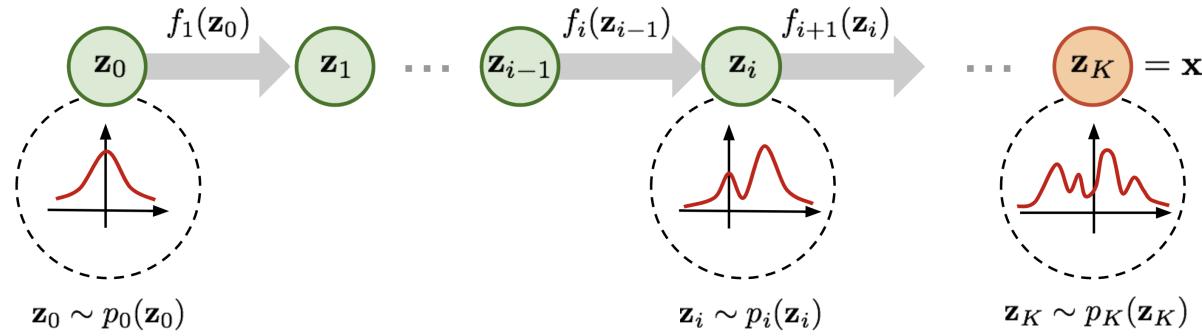
Hence, in order to determine the probability of x , we only need to determine its probability in latent space, and get the derivate of f . Note that this is for a univariate distribution, and f is required to be invertible and smooth. For a multivariate case, the derivative becomes a Jacobian of which we need to take the determinant. As we usually use the log-likelihood as objective, we write the multivariate term with logarithms below:

$$\log p_x(\mathbf{x}) = \log p_z(f(\mathbf{x})) + \log \left| \det \frac{df(\mathbf{x})}{d\mathbf{x}} \right|$$

Although we now know how a normalizing flow obtains its likelihood, it might not be clear what a normalizing flow does intuitively. For this, we should look from the inverse perspective of the flow starting with the prior probability density $p_z(z)$. If we apply an invertible function on it, we effectively “transform” its probability density. For instance, if $f^{-1}(z) = z + 1$, we shift the density by one while still remaining a valid probability distribution, and being invertible. We can also apply more complex transformations, like scaling: $f^{-1}(z) = 2z + 1$, but there you might see a difference. When you scale, you also change the volume of the probability density, as for example on uniform distributions (figure credit - [Eric Jang](#)):



You can see that the height of $p(y)$ should be lower than $p(x)$ after scaling. This change in volume represents $\left| \frac{df(x)}{dx} \right|$ in our equation above, and ensures that even after scaling, we still have a valid probability distribution. We can go on with making our function f more complex. However, the more complex f becomes, the harder it will be to find the inverse f^{-1} of it, and to calculate the log-determinant of the Jacobian $\log \left| \det \frac{df(\mathbf{x})}{d\mathbf{x}} \right|$. An easier trick to stack multiple invertible functions f_1, \dots, f_K after each other, as all together, they still represent a single, invertible function. Using multiple, learnable invertible functions, a normalizing flow attempts to transform $p_z(z)$ slowly into a more complex distribution which should finally be $p_x(x)$. We visualize the idea below (figure credit - [Lilian Weng](#)):



Starting from z_0 , which follows the prior Gaussian distribution, we sequentially apply the invertible functions f_1, f_2, \dots, f_K , until z_K represents x . Note that in the figure above, the functions f represent the inverted function from f we had above (here: $f : Z \rightarrow X$, above: $f : X \rightarrow Z$). This is just a different notation and has no impact on the actual flow design because all f need to be invertible anyways. When we estimate the log likelihood of a data point x as in the equations above, we run the flows in the opposite direction than visualized above. Multiple flow layers have been proposed that use a neural network as learnable parameters, such as the planar and radial flow. However, we will focus here on flows that are commonly used in image modeling, and will discuss them in the rest of the notebook along with the details of how to train a normalizing flow.

4.26.2 Normalizing Flows on images

To become familiar with normalizing flows, especially for the application of image modeling, it is best to discuss the different elements in a flow along with the implementation. As a general concept, we want to build a normalizing flow that maps an input image (here MNIST) to an equally sized latent space:

As a first step, we will implement a template of a normalizing flow. During training and validation, a normalizing flow performs density estimation in the forward direction. For this, we apply a series of flow transformations on the input x and estimate the probability of the input by determining the probability of the transformed point z given a prior, and the change of volume caused by the transformations. During inference, we can do both density estimation and sampling new points by inverting the flow transformations. Therefore, we define a function `_get_likelihood` which performs density estimation, and `sample` to generate new examples.

The standard metric used in generative models, and in particular normalizing flows, is bits per dimensions (bpd). Bpd is motivated from an information theory perspective and describes how many bits we would need to encode a particular example in our modeled distribution. The less bits we need, the more likely the example in our distribution. When we test for the bits per dimension of our test dataset, we can judge whether our model generalizes to new samples of the dataset and didn't memorize the training dataset. In order to calculate the bits per dimension score, we can rely on the negative log-likelihood and change the log base (as bits are binary while NLL is usually exponential):

$$\text{bpd} = \text{nll} \cdot \log_2(\exp(1)) \cdot \left(\prod d_i \right)^{-1}$$

where d_1, \dots, d_K are the dimensions of the input. For images, this would be the height, width and channel number. We divide the log likelihood by these extra dimensions to have a metric which we can compare for different image resolutions. In the original image space, MNIST examples have a bits per dimension score of 8 (we need 8 bits to encode each pixel as there are 256 possible values).

```
[5]: class ImageFlow(nn.Module):
    flows : Sequence[nn.Module] # A list of flows (each a nn.Module) that should be applied on the images.
    import_samples : int = 8 # Number of importance samples to use during testing (see explanation below).
    (continues on next page)
```

(continued from previous page)

```

def __call__(self, x, rng, testing=False):
    if not testing:
        bpd, rng = self._get_likelihood(x, rng)
    else:
        # Perform importance sampling during testing => estimate likelihood M times.
        # for each image
        img_ll, rng = self._get_likelihood(x.repeat(self.import_samples, 0),
                                           rng,
                                           return_ll=True)
        img_ll = img_ll.reshape(-1, self.import_samples)

        # To average the probabilities, we need to go from log-space to exp, and
        # back to log.
        # Logsumexp provides us a stable implementation for this
        img_ll = jax.nn.logsumexp(img_ll, axis=-1) - np.log(self.import_samples)

        # Calculate final bpd
        bpd = -img_ll * np.log2(np.exp(1)) / np.prod(x.shape[1:])
        bpd = bpd.mean()
    return bpd, rng

def encode(self, imgs, rng):
    # Given a batch of images, return the latent representation z and ldj of the
    # transformations
    z, ldj = imgs, jnp.zeros(imgs.shape[0])
    for flow in self.flows:
        z, ldj, rng = flow(z, ldj, rng, reverse=False)
    return z, ldj, rng

def _get_likelihood(self, imgs, rng, return_ll=False):
    """
    Given a batch of images, return the likelihood of those.
    If return_ll is True, this function returns the log likelihood of the input.
    Otherwise, the ouput metric is bits per dimension (scaled negative log_
    likelihood)
    """
    z, ldj, rng = self.encode(imgs, rng)
    log_pz = jax.scipy.stats.norm.logpdf(z).sum(axis=(1, 2, 3))
    log_px = ldj + log_pz
    nll = -log_px
    # Calculating bits per dimension
    bpd = nll * np.log2(np.exp(1)) / np.prod(imgs.shape[1:])
    return (bpd.mean() if not return_ll else log_px), rng

def sample(self, img_shape, rng, z_init=None):
    """
    Sample a batch of images from the flow.
    """
    # Sample latent representation from prior
    if z_init is None:
        rng, normal_rng = random.split(rng)

```

(continues on next page)

(continued from previous page)

```

z = random.normal(normal_rng, shape=img_shape)
else:
    z = z_init

# Transform z to x by inverting the flows
ldj = jnp.zeros(img_shape[0])
for flow in reversed(self.flows):
    z, ldj, rng = flow(z, ldj, rng, reverse=True)
return z, rng

```

The test step differs from the training and validation step in that it makes use of importance sampling. We will discuss the motivation and details behind this after understanding how flows model discrete images in continuous space.

Dequantization

Normalizing flows rely on the rule of change of variables, which is naturally defined in continuous space. Applying flows directly on discrete data leads to undesired density models where arbitrarily high likelihood are placed on a few, particular values. See the illustration below:

The black points represent the discrete points, and the green volume the density modeled by a normalizing flow in continuous space. The flow would continue to increase the likelihood for $x = 0, 1, 2, 3$ while having no volume on any other point. Remember that in continuous space, we have the constraint that the overall volume of the probability density must be 1 ($\int p(x)dx = 1$). Otherwise, we don't model a probability distribution anymore. However, the discrete points $x = 0, 1, 2, 3$ represent delta peaks with no width in continuous space. This is why the flow can place an infinite high likelihood on these few points while still representing a distribution in continuous space. Nonetheless, the learned density does not tell us anything about the distribution among the discrete points, as in discrete space, the likelihoods of those four points would have to sum to 1, not to infinity.

To prevent such degenerated solutions, a common solution is to add a small amount of noise to each discrete value, which is also referred to as dequantization. Considering x as an integer (as it is the case for images), the dequantized representation v can be formulated as $v = x + u$ where $u \in [0, 1]^D$. Thus, the discrete value 1 is modeled by a distribution over the interval $[1.0, 2.0]$, the value 2 by an volume over $[2.0, 3.0]$, etc. Our objective of modeling $p(x)$ becomes:

$$p(x) = \int p(x+u)du = \int \frac{q(u|x)}{q(u|x)} p(x+u)du = \mathbb{E}_{u \sim q(u|x)} \left[\frac{p(x+u)}{q(u|x)} \right]$$

with $q(u|x)$ being the noise distribution. For now, we assume it to be uniform, which can also be written as $p(x) = \mathbb{E}_{u \sim U(0,1)^D} [p(x+u)]$.

In the following, we will implement Dequantization as a flow transformation itself. After adding noise to the discrete values, we additionally transform the volume into a Gaussian-like shape. This is done by scaling $x + u$ between 0 and 1, and applying the invert of the sigmoid function $\sigma(z)^{-1} = \log z - \log 1 - z$. If we would not do this, we would face two problems:

1. The input is scaled between 0 and 256 while the prior distribution is a Gaussian with mean 0 and standard deviation 1. In the first iterations after initializing the parameters of the flow, we would have extremely low likelihoods for large values like 256. This would cause the training to diverge instantaneously.
2. As the output distribution is a Gaussian, it is beneficial for the flow to have a similarly shaped input distribution. This will reduce the modeling complexity that is required by the flow.

Overall, we can implement dequantization as follows:

```
[6]: class Dequantization(nn.Module):
    alpha : float = 1e-5 # Small constant that is used to scale the original input for
    ↪numerical stability.
    quants : int = 256    # Number of possible discrete values (usually 256 for 8-bit
    ↪image)

    def __call__(self, z, ldj, rng, reverse=False):
        if not reverse:
            z, ldj, rng = self.dequant(z, ldj, rng)
            z, ldj = self.sigmoid(z, ldj, reverse=True)
        else:
            z, ldj = self.sigmoid(z, ldj, reverse=False)
            z = z * self.quants
            ldj += np.log(self.quants) * np.prod(z.shape[1:])
            z = jnp.floor(z)
            z = jax.lax.clamp(min=0., x=z, max=self.quants-1.).astype(jnp.int32)
        return z, ldj, rng

    def sigmoid(self, z, ldj, reverse=False):
        # Applies an invertible sigmoid transformation
        if not reverse:
            ldj += (-z-2*jax.nn.softplus(-z)).sum(axis=[1,2,3])
            z = nn.sigmoid(z)
            # Reversing scaling for numerical stability
            ldj -= np.log(1 - self.alpha) * np.prod(z.shape[1:])
            z = (z - 0.5 * self.alpha) / (1 - self.alpha)
        else:
            z = z * (1 - self.alpha) + 0.5 * self.alpha # Scale to prevent boundaries 0
        ↪and 1
            ldj += np.log(1 - self.alpha) * np.prod(z.shape[1:])
            ldj += (-jnp.log(z) - jnp.log(1-z)).sum(axis=[1,2,3])
            z = jnp.log(z) - jnp.log(1-z)
        return z, ldj

    def dequant(self, z, ldj, rng):
        # Transform discrete values to continuous volumes
        z = z.astype(jnp.float32)
        rng, uniform_rng = random.split(rng)
        z = z + random.uniform(uniform_rng, z.shape)
        z = z / self.quants
        ldj -= np.log(self.quants) * np.prod(z.shape[1:])
        return z, ldj, rng
```

A good check whether a flow is correctly implemented or not, is to verify that it is invertible. Hence, we will dequantize a randomly chosen training image, and then quantize it again. We would expect that we would get the exact same image out:

```
[7]: ## Testing invertibility of dequantization layer
orig_img = train_set[0][0][None] # Example image
ldj = jnp.zeros(1,)
dequant_module = Dequantization()
dequant_rng = random.PRNGKey(5)
deq_img, ldj, dequant_rng = dequant_module(orig_img, ldj, dequant_rng, reverse=False)
```

(continues on next page)

(continued from previous page)

```
reconst_img, ldj, dequant_rng = dequant_module(deq_img, ldj, dequant_rng, reverse=True)
```

```
d1, d2 = jnp.where(orig_img.squeeze() != reconst_img.squeeze())
if len(d1) != 0:
    print("Dequantization was not invertible.")
    for i in range(d1.shape[0]):
        print("Original value:", orig_img[0,d1[i],d2[i],0].item())
        print("Reconstructed value:", reconst_img[0,d1[i],d2[i],0].item())
else:
    print("Successfully inverted dequantization")
```

```
# Layer is not strictly invertible due to float precision constraints
# assert (orig_img == reconst_img).all().item()
```

```
Successfully inverted dequantization
```

The test succeeds as we would expect. However, there is a chance that the test fails due to numerical inaccuracies in the sigmoid invert. While the input space to the inverted sigmoid is scaled between 0 and 1, the output space is between $-\infty$ and ∞ . And as we use 32 bits to represent the numbers (in addition to applying logs over and over again), such inaccuracies can occur and should not be worrisome. Nevertheless, it is good to be aware of them, and can be improved by using a double tensor (float64).

Finally, we can take our dequantization and actually visualize the distribution it transforms the discrete values into:

```
[8]: def visualize_dequantization(quants, prior=None):
    """
    Function for visualizing the dequantization values of discrete values in continuous_
    space
    """
    # Prior over discrete values. If not given, a uniform is assumed
    if prior is None:
        prior = np.ones(quants, dtype=np.float32) / quants
    prior = prior / prior.sum() # Ensure proper categorical distribution

    inp = jnp.arange(-4, 4, 0.01).reshape(-1, 1, 1, 1) # Possible continuous values we_
    want to consider
    ldj = jnp.zeros(inp.shape[0])
    dequant_module = Dequantization(quants=quants)
    # Invert dequantization on continuous values to find corresponding discrete value
    out, ldj, _ = dequant_module(inp, ldj, rng=None, reverse=True)
    inp, out, prob = inp.squeeze(), out.squeeze(), jnp.exp(ldj)
    prob = prob * prior[out] # Probability scaled by categorical prior

    # Plot volumes and continuous distribution
    sns.set_style("white")
    fig = plt.figure(figsize=(6,3))
    x_ticks = []
    for v in np.unique(out):
        indices = np.where(out==v)
        color = to_rgb(f"C{v}")
        plt.fill_between(inp[indices], prob[indices], np.zeros(indices[0].shape[0]),_
        color=color+(0.5,), label=str(v))
        plt.plot([inp[indices[0][0]]]*2, [0, prob[indices[0][0]]], color=color)
```

(continues on next page)

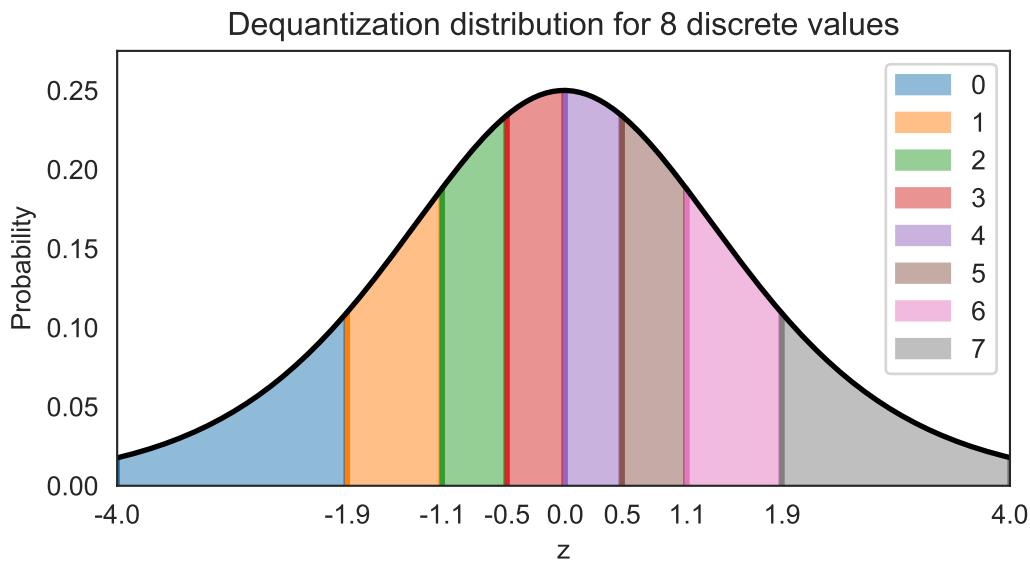
(continued from previous page)

```

plt.plot([inp[indices[0][-1]]]*2, [0, prob[indices[0][-1]]], color=color)
x_ticks.append(inp[indices[0][0]])
x_ticks.append(inp.max())
plt.xticks(x_ticks, [f'{x:.1f}' for x in x_ticks])
plt.plot(inp, prob, color=(0.0,0.0,0.0))
# Set final plot properties
plt.ylim(0, prob.max()*1.1)
plt.xlim(inp.min(), inp.max())
plt.xlabel("z")
plt.ylabel("Probability")
plt.title(f"Dequantization distribution for {quants} discrete values")
plt.legend()
plt.show()
plt.close()

visualize_dequantization(quants=8)

```



The visualized distribution show the sub-volumes that are assigned to the different discrete values. The value 0 has its volume between $[-\infty, -1.9]$, the value 1 is represented by the interval $[-1.9, -1.1]$, etc. The volume for each discrete value has the same probability mass. That's why the volumes close to the center (e.g. 3 and 4) have a smaller area on the z-axis as others (z is being used to denote the output of the whole dequantization flow).

Effectively, the consecutive normalizing flow models discrete images by the following objective:

$$\log p(x) = \log \mathbb{E}_{u \sim q(u|x)} \left[\frac{p(x+u)}{q(u|x)} \right] \geq \mathbb{E}_u \left[\log \frac{p(x+u)}{q(u|x)} \right]$$

Although normalizing flows are exact in likelihood, we have a lower bound. Specifically, this is an example of the Jensen inequality because we need to move the log into the expectation so we can use Monte-carlo estimates. In general, this bound is considerably smaller than the ELBO in variational autoencoders. Actually, we can reduce the bound ourselves by estimating the expectation not by one, but by M samples. In other words, we can apply importance sampling which leads to the following inequality:

$$\log p(x) = \log \mathbb{E}_{u \sim q(u|x)} \left[\frac{p(x+u)}{q(u|x)} \right] \geq \mathbb{E}_u \left[\log \frac{1}{M} \sum_{m=1}^M \frac{p(x+u_m)}{q(u_m|x)} \right] \geq \mathbb{E}_u \left[\log \frac{p(x+u)}{q(u|x)} \right]$$

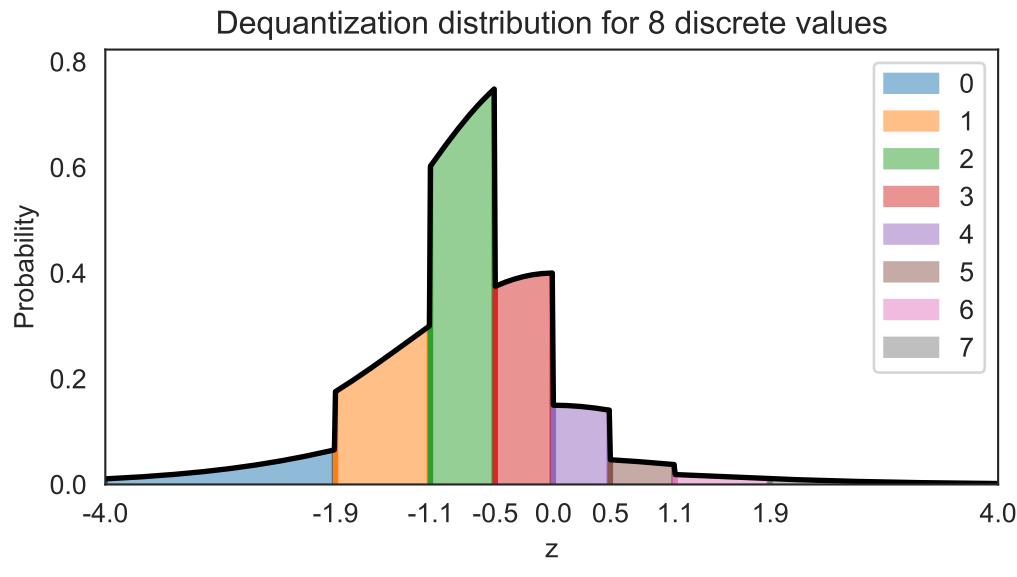
The importance sampling $\frac{1}{M} \sum_{m=1}^M \frac{p(x+u_m)}{q(u_m|x)}$ becomes $\mathbb{E}_{u \sim q(u|x)} \left[\frac{p(x+u)}{q(u|x)} \right]$ if $M \rightarrow \infty$, so that the more samples we use, the tighter the bound is. During testing, we can make use of this property and have it implemented in `test_step` in ImageFlow. In theory, we could also use this tighter bound during training. However, related work has shown that this does not necessarily lead to an improvement given the additional computational cost, and it is more efficient to stick with a single estimate [5].

Variational Dequantization

Dequantization uses a uniform distribution for the noise u which effectively leads to images being represented as hypercubes (cube in high dimensions) with sharp borders. However, modeling such sharp borders is not easy for a flow as it uses smooth transformations to convert it into a Gaussian distribution.

Another way of looking at it is if we change the prior distribution in the previous visualization. Imagine we have independent Gaussian noise on pixels which is commonly the case for any real-world taken picture. Therefore, the flow would have to model a distribution as above, but with the individual volumes scaled as follows:

```
[9]: visualize_dequantization(quants=8, prior=np.array([0.075, 0.2, 0.4, 0.2, 0.075, 0.025, 0.0125, 0.0125]))
```



Transforming such a probability into a Gaussian is a difficult task, especially with such hard borders. Dequantization has therefore been extended to more sophisticated, learnable distributions beyond uniform in a variational framework. In particular, if we remember the learning objective $\log p(x) = \log \mathbb{E}_u \left[\frac{p(x+u)}{q(u|x)} \right]$, the uniform distribution can be replaced by a learned distribution $q_\theta(u|x)$ with support over $u \in [0, 1]^D$. This approach is called Variational Dequantization and has been proposed by Ho et al. [3]. How can we learn such a distribution? We can use a second normalizing flow that takes x as external input and learns a flexible distribution over u . To ensure a support over $[0, 1]^D$, we can apply a sigmoid activation function as final flow transformation.

Inheriting the original dequantization class, we can implement variational dequantization as follows:

```
[10]: class VariationalDequantization(Dequantization):
    var_flows : Sequence[nn.Module] = None # A list of flow transformations to use for
    #modeling q(u/x)

    def dequant(self, z, ldj, rng):
```

(continues on next page)

(continued from previous page)

```

z = z.astype(jnp.float32)
img = (z / 255.0) * 2 - 1 # We condition the flows on x, i.e. the original image

# Prior of u is a uniform distribution as before
# As most flow transformations are defined on [-infinity,+infinity], we apply an
# inverse sigmoid first.
rng, uniform_rng = random.split(rng)
deq_noise = random.uniform(uniform_rng, z.shape)
deq_noise, ldj = self.sigmoid(deq_noise, ldj, reverse=True)
if self.var_flows is not None:
    for flow in self.var_flows:
        deq_noise, ldj, rng = flow(deq_noise, ldj, rng, reverse=False, orig_


```

Variational dequantization can be used as a substitute for dequantization. We will compare dequantization and variational dequantization in later experiments.

Coupling layers

Next, we look at possible transformations to apply inside the flow. A recent popular flow layer, which works well in combination with deep neural networks, is the coupling layer introduced by Dinh et al. [1]. The input z is arbitrarily split into two parts, $z_{1:j}$ and $z_{j+1:d}$, of which the first remains unchanged by the flow. Yet, $z_{1:j}$ is used to parameterize the transformation for the second part, $z_{j+1:d}$. Various transformations have been proposed in recent time [3,4], but here we will settle for the simplest and most efficient one: affine coupling. In this coupling layer, we apply an affine transformation by shifting the input by a bias μ and scale it by σ . In other words, our transformation looks as follows:

$$z'_{j+1:d} = \mu_\theta(z_{1:j}) + \sigma_\theta(z_{1:j}) \odot z_{j+1:d}$$

The functions μ and σ are implemented as a shared neural network, and the sum and multiplication are performed element-wise. The LDJ is thereby the sum of the logs of the scaling factors: $\sum_i [\log \sigma_\theta(z_{1:j})]_i$. Inverting the layer can as simply be done as subtracting the bias and dividing by the scale:

$$z_{j+1:d} = (z'_{j+1:d} - \mu_\theta(z_{1:j})) / \sigma_\theta(z_{1:j})$$

We can also visualize the coupling layer in form of a computation graph, where z_1 represents $z_{1:j}$, and z_2 represents $z_{j+1:d}$:

In our implementation, we will realize the splitting of variables as masking. The variables to be transformed, $z_{j+1:d}$, are masked when passing z to the shared network to predict the transformation parameters. When applying the transformation, we mask the parameters for $z_{1:j}$ so that we have an identity operation for those variables:

```
[11]: class CouplingLayer(nn.Module):
    network : nn.Module # NN to use in the flow for predicting mu and sigma
    mask : np.ndarray # Binary mask where 0 denotes that the element should be_
    #transformed, and 1 not.
```

(continues on next page)

(continued from previous page)

```

c_in : int # Number of input channels

def setup(self):
    self.scaling_factor = self.param('scaling_factor',
                                      nn.initializers.zeros,
                                      (self.c_in,))

def __call__(self, z, ldj, rng, reverse=False, orig_img=None):
    """
    Inputs:
        z - Latent input to the flow
        ldj - The current ldj of the previous flows.
              The ldj of this layer will be added to this tensor.
        rng - PRNG state
        reverse - If True, we apply the inverse of the layer.
        orig_img (optional) - Only needed in VarDeq. Allows external
                             input to condition the flow on (e.g. original image)
    """
    # Apply network to masked input
    z_in = z * self.mask
    if orig_img is None:
        nn_out = self.network(z_in)
    else:
        nn_out = self.network(jnp.concatenate([z_in, orig_img], axis=-1))
    s, t = nn_out.split(2, axis=-1)

    # Stabilize scaling output
    s_fac = jnp.exp(self.scaling_factor).reshape(1, 1, 1, -1)
    s = nn.tanh(s / s_fac) * s_fac

    # Mask outputs (only transform the second part)
    s = s * (1 - self.mask)
    t = t * (1 - self.mask)

    # Affine transformation
    if not reverse:
        # Whether we first shift and then scale, or the other way round,
        # is a design choice, and usually does not have a big impact
        z = (z + t) * jnp.exp(s)
        ldj += s.sum(axis=[1, 2, 3])
    else:
        z = (z * jnp.exp(-s)) - t
        ldj -= s.sum(axis=[1, 2, 3])

    return z, ldj, rng

```

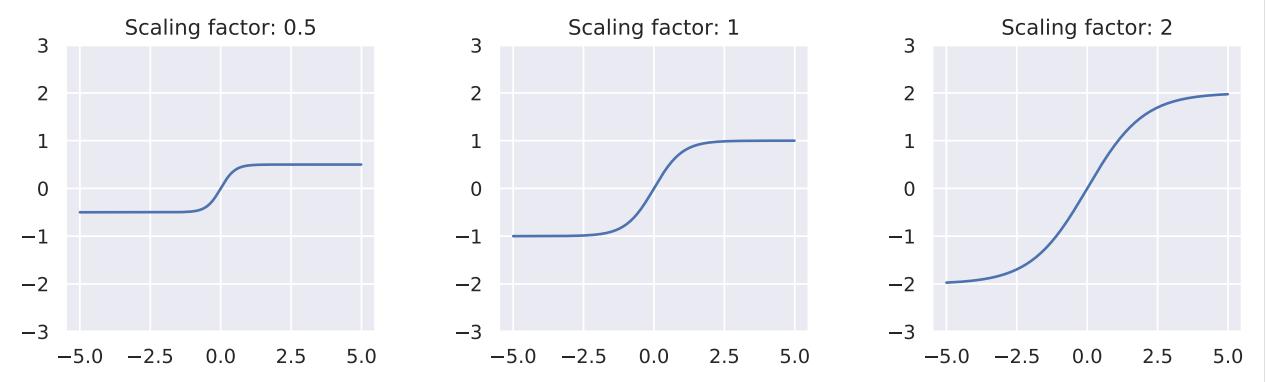
For stabilization purposes, we apply a tanh activation function on the scaling output. This prevents sudden large output values for the scaling that can destabilize training. To still allow scaling factors smaller or larger than -1 and 1 respectively, we have a learnable parameter per dimension, called `scaling_factor`. This scales the tanh to different limits. Below, we visualize the effect of the scaling factor on the output activation of the scaling terms:

[12]: `x = jnp.arange(-5, 5, 0.01)`

(continues on next page)

(continued from previous page)

```
scaling_factors = [0.5, 1, 2]
sns.set()
fig, ax = plt.subplots(1, 3, figsize=(12,3))
for i, scale in enumerate(scaling_factors):
    y = nn.tanh(x / scale) * scale
    ax[i].plot(x, y)
    ax[i].set_title("Scaling factor: " + str(scale))
    ax[i].set_ylim(-3, 3)
plt.subplots_adjust(wspace=0.4)
sns.reset_orig()
plt.show()
```



Coupling layers generalize to any masking technique we could think of. However, the most common approach for images is to split the input z in half, using a checkerboard mask or channel mask. A checkerboard mask splits the variables across the height and width dimensions and assigns each other pixel to $z_{j+1:d}$. Thereby, the mask is shared across channels. In contrast, the channel mask assigns half of the channels to $z_{j+1:d}$, and the other half to $z_{1:j+1}$. Note that when we apply multiple coupling layers, we invert the masking for each other layer so that each variable is transformed a similar amount of times.

Let's implement a function that creates a checkerboard mask and a channel mask for us:

```
[13]: def create_checkerboard_mask(h, w, invert=False):
    x, y = jnp.arange(h, dtype=jnp.int32), jnp.arange(w, dtype=jnp.int32)
    xx, yy = jnp.meshgrid(x, y, indexing='ij')
    mask = jnp.fmod(xx + yy, 2)
    mask = mask.astype(jnp.float32).reshape(1, h, w, 1)
    if invert:
        mask = 1 - mask
    return mask

def create_channel_mask(c_in, invert=False):
    mask = jnp.concatenate([
        jnp.ones((c_in//2,), dtype=jnp.float32),
        jnp.zeros((c_in-c_in//2,), dtype=jnp.float32)
    ])
    mask = mask.reshape(1, 1, 1, c_in)
    if invert:
        mask = 1 - mask
    return mask
```

We can also visualize the corresponding masks for an image of size $8 \times 8 \times 2$ (2 channels):

```
[14]: checkerboard_mask = create_checkerboard_mask(h=8, w=8).repeat(2, -1)
channel_mask = jnp.resize(create_channel_mask(c_in=2), (1, 8, 8, 2))
show_imgs(checkerboard_mask.swapaxes(0, 3), "Checkerboard mask")
show_imgs(channel_mask.swapaxes(0, 3), "Channel mask")
```

Checkerboard mask



Channel mask



As a last aspect of coupling layers, we need to decide for the deep neural network we want to apply in the coupling layers. The input to the layers is an image, and hence we stick with a CNN. Because the input to a transformation depends on all transformations before, it is crucial to ensure a good gradient flow through the CNN back to the input, which can be optimally achieved by a ResNet-like architecture. Specifically, we use a Gated ResNet that adds a σ -gate to the skip connection, similarly to the input gate in LSTMs. The details are not necessarily important here, and the network is strongly inspired from Flow++ [3] in case you are interested in building even stronger models.

```
[15]: class ConcatELU(nn.Module):
    """
    Activation function that applies ELU in both direction (inverted and plain).
    Allows non-linearity while providing strong gradients for any input (important for
    final convolution)
    """

    def __call__(self, x):
        return jnp.concatenate([nn.elu(x), nn.elu(-x)], axis=-1)

class GatedConv(nn.Module):
    """
    This module applies a two-layer convolutional ResNet block with input gate """
    c_in : int # Number of input channels
    c_hidden : int # Number of hidden dimensions

    @nn.compact
    def __call__(self, x):
        out = nn.Sequential([
            ConcatELU(),
            nn.Conv(self.c_hidden, kernel_size=(3, 3)),
```

(continues on next page)

(continued from previous page)

```

        ConcatELU(),
        nn.Conv(2*self.c_in, kernel_size=(1, 1))
    ])(x)
    val, gate = out.split(2, axis=-1)
    return x + val * nn.sigmoid(gate)

class GatedConvNet(nn.Module):
    c_hidden : int # Number of hidden dimensions to use within the network
    c_out : int # Number of output channels
    num_layers : int = 3 # Number of gated ResNet blocks to apply

    def setup(self):
        layers = []
        layers += [nn.Conv(self.c_hidden, kernel_size=(3, 3))]
        for layer_index in range(self.num_layers):
            layers += [GatedConv(self.c_hidden, self.c_hidden),
                       nn.LayerNorm()]
        layers += [ConcatELU(),
                   nn.Conv(self.c_out, kernel_size=(3, 3),
                          kernel_init=nn.initializers.zeros)]
        self.nn = nn.Sequential(layers)

    def __call__(self, x):
        return self.nn(x)

```

```
[16]: ## Test MultiheadAttention implementation
# Example features as input
main_rng, x_rng = random.split(main_rng)
x = random.normal(x_rng, (3, 32, 32, 16))
# Create attention
mh_attn = GatedConvNet(c_hidden=32, c_out=18, num_layers=3)
# Initialize parameters of attention with random key and inputs
main_rng, init_rng = random.split(main_rng)
params = mh_attn.init(init_rng, x)['params']
# Apply attention with parameters on the inputs
out = mh_attn.apply({'params': params}, x)
print('Out', out.shape)

del mh_attn, params
Out (3, 32, 32, 18)
```

Training loop

Finally, we can add Dequantization, Variational Dequantization and Coupling Layers together to build our full normalizing flow on MNIST images. We apply 8 coupling layers in the main flow, and 4 for variational dequantization if applied. We apply a checkerboard mask throughout the network as with a single channel (black-white images), we cannot apply channel mask. The overall architecture is visualized below.

```
[17]: def create_simple_flow(use_vardeq=True):
    flow_layers = []
    if use_vardeq:
        vardeq_layers = [CouplingLayer(network=GatedConvNet(c_out=2, c_hidden=16),
                                         mask=create_checkerboard_mask(h=28, w=28,
                                         invert=(i%2==1)),
                           c_in=1) for i in range(4)]
        flow_layers += [VariationalDequantization(var_flows=vardeq_layers)]
    else:
        flow_layers += [Dequantization()]

    for i in range(8):
        flow_layers += [CouplingLayer(network=GatedConvNet(c_out=2, c_hidden=32),
                                         mask=create_checkerboard_mask(h=28, w=28, invert=(i%2==1)),
                           c_in=1)]

    flow_model = ImageFlow(flow_layers)
    return flow_model
```

For implementing the training loop, we use a similar trainer module as we have done in several tutorials before. Note that we again provide pre-trained models (see later on in the notebook) as normalizing flows are particularly expensive to train. We have also run validation and testing as this can take some time as well with the added importance sampling.

```
[18]: class TrainerModule:

    def __init__(self, model_name, flow, lr=1e-3, seed=42):
        super().__init__()
        self.model_name = model_name
        self.lr = lr
        self.seed = seed
        # Create empty model. Note: no parameters yet
        self.model = flow
        # Prepare logging
        self.exmp_imgs = next(iter(train_exmp_loader))[0]
        self.log_dir = os.path.join(CHECKPOINT_PATH, self.model_name)
        self.logger = SummaryWriter(log_dir=self.log_dir)
        # Create jitted training and eval functions
        self.create_functions()
        # Initialize model
        self.init_model()

    def create_functions(self):
        # Training function
        def train_step(state, rng, batch):
```

(continues on next page)

(continued from previous page)

```

        imgs, _ = batch
        loss_fn = lambda params: self.model.apply({'params': params}, imgs, rng, ↵
→ testing=False)
        (loss, rng), grads = jax.value_and_grad(loss_fn, has_aux=True)(state.params) ↵
→ # Get loss and gradients for loss
        state = state.apply_gradients(grads=grads) # Optimizer update step
        return state, rng, loss
    self.train_step = jax.jit(train_step)
    # Eval function, which is separately jitted for validation and testing
    def eval_step(state, rng, batch, testing):
        return self.model.apply({'params': state.params}, batch[0], rng, ↵
→ testing=testing)
        self.eval_step = jax.jit(eval_step, static_argnums=(3,))

    def init_model(self):
        # Initialize model
        self.rng = jax.random.PRNGKey(self.seed)
        self.rng, init_rng, flow_rng = jax.random.split(self.rng, 3)
        params = self.model.init(init_rng, self.exmp_imgs, flow_rng)['params']
        # Initialize learning rate schedule and optimizer
        lr_schedule = optax.exponential_decay(
            init_value=self.lr,
            transition_steps=len(train_data_loader),
            decay_rate=0.99,
            end_value=0.01*self.lr
        )
        optimizer = optax.chain(
            optax.clip_by_global_norm(1.0), # Clip gradients at 1
            optax.adam(lr_schedule)
        )
        # Initialize training state
        self.state = train_state.TrainState.create(apply_fn=self.model.apply, ↵
→ params=params, tx=optimizer)

    def train_model(self, train_loader, val_loader, num_epochs=500):
        # Train model for defined number of epochs
        best_eval = 1e6
        for epoch_idx in tqdm(range(1, num_epochs+1)):
            self.train_epoch(train_loader, epoch=epoch_idx)
            if epoch_idx % 5 == 0:
                eval_bpd = self.eval_model(val_loader, testing=False)
                self.logger.add_scalar('val/bpd', eval_bpd, global_step=epoch_idx)
                if eval_bpd < best_eval:
                    best_eval = eval_bpd
                    self.save_model(step=epoch_idx)
            self.logger.flush()

    def train_epoch(self, data_loader, epoch):
        # Train model for one epoch, and log avg loss
        avg_loss = 0.
        for batch in tqdm(data_loader, leave=False):
            self.state, self.rng, loss = self.train_step(self.state, self.rng, batch)

```

(continues on next page)

(continued from previous page)

```

        avg_loss += loss
    avg_loss /= len(data_loader)
    self.logger.add_scalar('train/bpd', avg_loss.item(), global_step=epoch)

def eval_model(self, data_loader, testing=False):
    # Test model on all images of a data loader and return avg loss
    losses = []
    batch_sizes = []
    for batch in data_loader:
        loss, self.rng = self.eval_step(self.state, self.rng, batch, testing=testing)
        losses.append(loss)
        batch_sizes.append(batch[0].shape[0])
    losses_np = np.stack(jax.device_get(losses))
    batch_sizes_np = np.stack(batch_sizes)
    avg_loss = (losses_np * batch_sizes_np).sum() / batch_sizes_np.sum()
    return avg_loss

def save_model(self, step=0):
    # Save current model at certain training iteration
    checkpoints.save_checkpoint(ckpt_dir=self.log_dir, target=self.state.params, ↴
→step=step)

def load_model(self, pretrained=False):
    # Load model. We use different checkpoint for pretrained models
    if not pretrained:
        params = checkpoints.restore_checkpoint(ckpt_dir=self.log_dir, target=self. ↴
→state.params)
    else:
        params = checkpoints.restore_checkpoint(ckpt_dir=os.path.join(CHECKPOINT_ ↴
→PATH, f'{self.model_name}.ckpt'), target=self.state.params)
        self.state = train_state.TrainState.create(apply_fn=self.model.apply, ↴
→params=params, tx=self.state.tx)

def checkpoint_exists(self):
    # Check whether a pretrained model exist for this autoencoder
    return os.path.isfile(os.path.join(CHECKPOINT_PATH, f'{self.model_name}.ckpt'))
```

```
[19]: def train_flow(flow, model_name="MNISTFlow"):
    # Create a trainer module with specified hyperparameters
    trainer = TrainerModule(model_name, flow)
    if not trainer.checkpoint_exists(): # Skip training if pretrained model exists
        trainer.train_model(train_data_loader,
                            val_loader,
                            num_epochs=200)
    trainer.load_model()
    val_bpd = trainer.eval_model(val_loader, testing=True)
    start_time = time.time()
    test_bpd = trainer.eval_model(test_loader, testing=True)
    duration = time.time() - start_time
    results = {'val': val_bpd, 'test': test_bpd, 'time': duration / len(test_loader), ↴
→/ trainer.model.import_samples}
```

(continues on next page)

(continued from previous page)

```

else:
    trainer.load_model(pretrained=True)
    with open(os.path.join(CHECKPOINT_PATH, f'{trainer.model_name}_results.json'), 'r') as f:
        results = json.load(f)

    # Bind parameters to model for easier inference
    trainer.model_bd = trainer.model.bind({'params': trainer.state.params})
return trainer, results

```

4.26.3 Multi-scale architecture

One disadvantage of normalizing flows is that they operate on the exact same dimensions as the input. If the input is high-dimensional, so is the latent space, which requires larger computational cost to learn suitable transformations. However, particularly in the image domain, many pixels contain less information in the sense that we could remove them without loosing the semantical information of the image.

Based on this intuition, deep normalizing flows on images commonly apply a multi-scale architecture [1]. After the first N flow transformations, we split off half of the latent dimensions and directly evaluate them on the prior. The other half is run through N more flow transformations, and depending on the size of the input, we split it again in half or stop overall at this position. The two operations involved in this setup is `Squeeze` and `Split` which we will review more closely and implement below.

Squeeze and Split

When we want to remove half of the pixels in an image, we have the problem of deciding which variables to cut, and how to rearrange the image. Thus, the squeezing operation is commonly used before split, which divides the image into subsquares of shape $2 \times 2 \times C$, and reshapes them into $1 \times 1 \times 4C$ blocks. Effectively, we reduce the height and width of the image by a factor of 2 while scaling the number of channels by 4. Afterwards, we can perform the split operation over channels without the need of rearranging the pixels. The smaller scale also makes the overall architecture more efficient. Visually, the squeeze operation should transform the input as follows:

The input of $4 \times 4 \times 1$ is scaled to $2 \times 2 \times 4$ following the idea of grouping the pixels in $2 \times 2 \times 1$ subsquares. Next, let's try to implement this layer:

```
[20]: class SqueezeFlow(nn.Module):

    def __call__(self, z, ldj, rng, reverse=False):
        B, H, W, C = z.shape
        if not reverse:
            # Forward direction: H x W x C => H/2 x W/2 x 4C
            z = z.reshape(B, H//2, 2, W//2, 2, C)
            z = z.transpose((0, 1, 3, 2, 4, 5))
            z = z.reshape(B, H//2, W//2, 4*C)
        else:
            # Reverse direction: H/2 x W/2 x 4C => H x W x C
            z = z.reshape(B, H, W, 2, 2, C//4)
            z = z.transpose((0, 1, 3, 2, 4, 5))
            z = z.reshape(B, H*2, W*2, C//4)
        return z, ldj, rng
```

Before moving on, we can verify our implementation by comparing our output with the example figure above:

```
[21]: sq_flow = SqueezeFlow()
rand_img = jnp.arange(1, 17).reshape(1, 4, 4, 1)
print("Image (before)\n", rand_img.transpose(0, 3, 1, 2)) # Permute for readability
forward_img, _, _ = sq_flow(rand_img, ldj=None, rng=None, reverse=False)
print("\nImage (forward)\n", forward_img)
reconst_img, _, _ = sq_flow(forward_img, ldj=None, rng=None, reverse=True)
print("\nImage (reverse)\n", reconst_img.transpose(0, 3, 1, 2))

Image (before)
[[[ [ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]
  [13 14 15 16]]]

Image (forward)
[[[ [ 1  2  5  6]
  [ 3  4  7  8]
  [[ 9 10 13 14]
  [11 12 15 16]]]

Image (reverse)
[[[ [ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]
  [13 14 15 16]]]
```

The split operation divides the input into two parts, and evaluates one part directly on the prior. So that our flow operation fits to the implementation of the previous layers, we will return the prior probability of the first part as the log determinant jacobian of the layer. It has the same effect as if we would combine all variable splits at the end of the flow, and evaluate them together on the prior.

```
[22]: class SplitFlow(nn.Module):

    def __call__(self, z, ldj, rng, reverse=False):
        if not reverse:
            z, z_split = z.split(2, axis=-1)
            ldj += jax.scipy.stats.norm.logpdf(z_split).sum(axis=[1,2,3])
        else:
            z_split = random.normal(rng, z.shape)
            z = jnp.concatenate([z, z_split], axis=-1)
            ldj -= jax.scipy.stats.norm.logpdf(z_split).sum(axis=[1,2,3])
        return z, ldj, rng
```

Building a multi-scale flow

After defining the squeeze and split operation, we are finally able to build our own multi-scale flow. Deep normalizing flows such as Glow and Flow++ [2,3] often apply a split operation directly after squeezing. However, with shallow flows, we need to be more thoughtful about where to place the split operation as we need at least a minimum amount of transformations on each variable. Our setup is inspired by the original RealNVP architecture [1] which is shallower than other, more recent state-of-the-art architectures.

Hence, for the MNIST dataset, we will apply the first squeeze operation after two coupling layers, but don't apply a split operation yet. Because we have only used two coupling layers and each the variable has been only transformed once, a split operation would be too early. We apply two more coupling layers before finally applying a split flow and squeeze again. The last four coupling layers operate on a scale of $7 \times 7 \times 8$. The full flow architecture is shown below.

Note that while the feature maps inside the coupling layers reduce with the height and width of the input, the increased number of channels is not directly considered. To counteract this, we increase the hidden dimensions for the coupling layers on the squeezed input. The dimensions are often scaled by 2 as this approximately increases the computation cost by 4 canceling with the squeezing operation. However, we will choose the hidden dimensionalities 32, 48, 64 for the three scales respectively to keep the number of parameters reasonable and show the efficiency of multi-scale architectures.

```
[23]: def create_multiscale_flow():
    flow_layers = []

    vardeq_layers = [CouplingLayer(network=GatedConvNet(c_out=2, c_hidden=16),
                                    mask=create_checkerboard_mask(h=28, w=28, invert=(i%2==1)),
                     c_in=1) for i in range(4)]
    flow_layers += [VariationalDequantization(var_flows=vardeq_layers)]

    flow_layers += [CouplingLayer(network=GatedConvNet(c_out=2, c_hidden=32),
                                    mask=create_checkerboard_mask(h=28, w=28, invert=(i%2==1)),
                     c_in=1) for i in range(2)]
    flow_layers += [SqueezeFlow()]
    for i in range(2):
        flow_layers += [CouplingLayer(network=GatedConvNet(c_out=8, c_hidden=48),
                                    mask=create_channel_mask(c_in=4, invert=(i%2==1)),
                     c_in=4)]
    flow_layers += [SplitFlow(),
                  SqueezeFlow()]
    for i in range(4):
        flow_layers += [CouplingLayer(network=GatedConvNet(c_out=16, c_hidden=64),
                                    mask=create_channel_mask(c_in=8, invert=(i%2==1)),
                     c_in=8)]

    flow_model = ImageFlow(flow_layers)
    return flow_model
```

4.26.4 Analysing the flows

In the last part of the notebook, we will train all the models we have implemented above, and try to analyze the effect of the multi-scale architecture and variational dequantization.

Training flow variants

Before we can analyse the flow models, we need to train them first. We provide pre-trained models that contain the validation and test performance, and run-time information. As flow models are computationally expensive, we advice you to rely on those pretrained models for a first run through the notebook.

```
[24]: flow_dict = {"simple": {}, "vardeq": {}, "multiscale": {}}
flow_dict["simple"]["model"], flow_dict["simple"]["result"] = train_flow(create_simple_
˓→flow(use_vardeq=False), model_name="MNISTFlow_simple")
flow_dict["vardeq"]["model"], flow_dict["vardeq"]["result"] = train_flow(create_simple_
˓→flow(use_vardeq=True), model_name="MNISTFlow_vardeq")
flow_dict["multiscale"]["model"], flow_dict["multiscale"]["result"] = train_flow(create_
˓→multiscale_flow(), model_name="MNISTFlow_multiscale")
```

We can show the difference in number of parameters below:

```
[25]: def print_num_params(model):
    num_params = sum([np.prod(p.shape) for p in jax.tree_util.tree_leaves(model.state.
→params)])
    print("Number of parameters: {:.3f}".format(num_params))

print_num_params(flow_dict["simple"]["model"])
print_num_params(flow_dict["vardeq"]["model"])
print_num_params(flow_dict["multiscale"]["model"])

Number of parameters: 556,312
Number of parameters: 628,388
Number of parameters: 1,711,818
```

Although the multi-scale flow has almost 3 times the parameters of the single scale flow, it is not necessarily more computationally expensive than its counterpart. We will compare the runtime in the following experiments as well.

Density modeling and sampling

Firstly, we can compare the models on their quantitative results. The following table shows all important statistics. The inference time specifies the time needed to determine the probability for a batch of 64 images for each model, and the sampling time the duration it took to sample a batch of 64 images.

```
[26]: %html
<!-- Some HTML code to increase font size in the following table -->
<style>
th {font-size: 120%;}
td {font-size: 120%;}
</style>
<IPython.core.display.HTML object>
```

```
[27]: import tabulate
from IPython.display import display, HTML

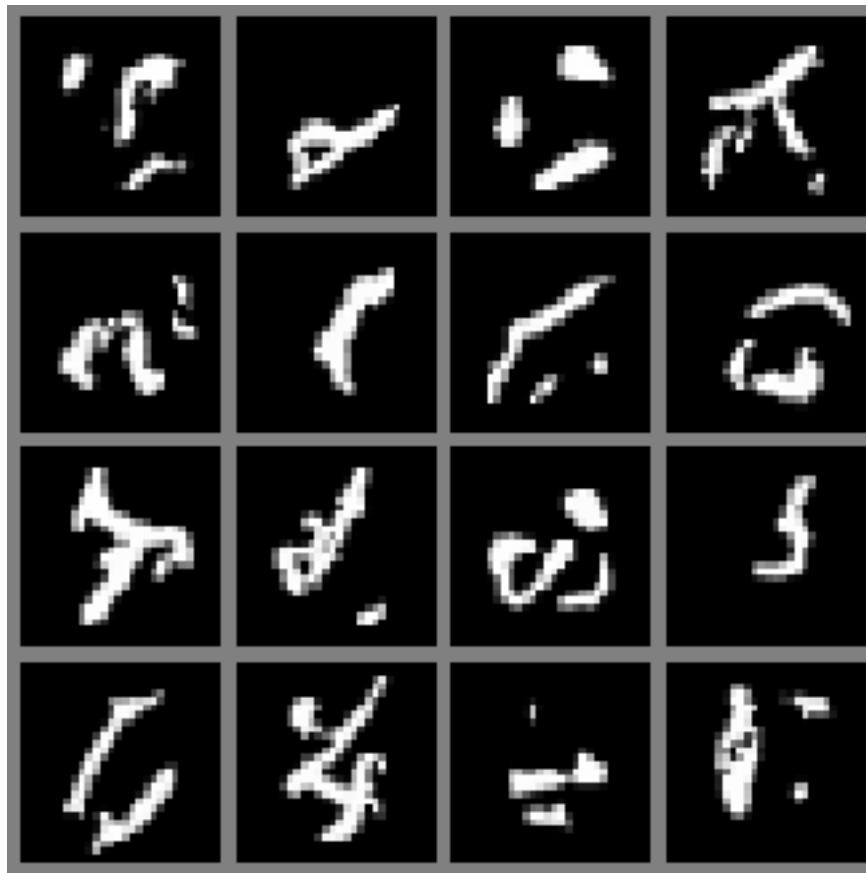
table = [[key,
          "%4.3f bpd" % flow_dict[key]["result"]["val"],
          "%4.3f bpd" % flow_dict[key]["result"]["test"],
          "%2.1f ms" % (1000 * flow_dict[key]["result"]["time"]),
          "%2.1f ms" % (1000 * flow_dict[key]["result"].get("samp_time", 0)),
          "{}:{}".format(sum([np.prod(p.shape) for p in jax.tree_util.tree_leaves(flow_
dict[key]["model"].state.params)]))]
         for key in flow_dict]
display(HTML(tabulate.tabulate(table, tablefmt='html', headers=["Model", "Validation Bpd
", "Test Bpd", "Inference time", "Sampling time", "Num Parameters"])))
<IPython.core.display.HTML object>
```

As we have initially expected, using variational dequantization improves upon standard dequantization in terms of bits per dimension. Although the difference with 0.04bpd doesn't seem impressive first, it is a considerably step for generative models (most state-of-the-art models improve upon previous models in a range of 0.02-0.1bpd on CIFAR with three times as high bpd). While it takes longer to evaluate the probability of an image due to the variational dequantization, which also leads to a longer training time, it does not have an effect on the sampling time. This is because inverting variational dequantization is the same as dequantization: finding the next lower integer.

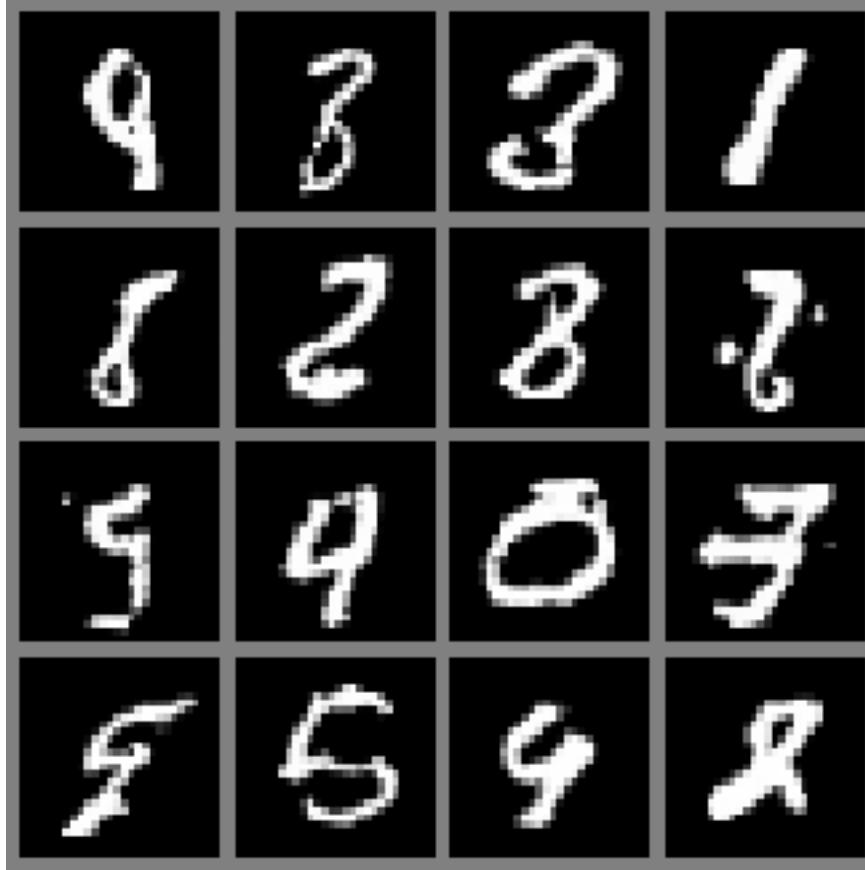
When we compare the two models to multi-scale architecture, we can see that the bits per dimension score again dropped by about 0.02bpd. Additionally, the inference time and sampling time improved notably despite having more parameters. Thus, we see that the multi-scale flow is not only stronger for density modeling, but also more efficient.

Next, we can test the sampling quality of the models. We should note that the samples for variational dequantization and standard dequantization are very similar, and hence we visualize here only the ones for variational dequantization and the multi-scale model. However, feel free to also test out the "simple" model. The seeds are set to obtain reproducible generations and are not cherry picked.

```
[28]: sample_rng = random.PRNGKey(44)
samples, _ = flow_dict["vardeq"]["model"].model_bd.sample(img_shape=[16, 28, 28, 1],_
rng=sample_rng)
show_imgs(samples)
```



```
[29]: sample_rng = random.PRNGKey(44)
samples, _ = flow_dict["multiscale"]["model"].model_bd.sample(img_shape=[16, 7, 7, 8],_
rng=sample_rng)
show_imgs(samples)
```



From the few samples, we can see a clear difference between the simple and the multi-scale model. The single-scale model has only learned local, small correlations while the multi-scale model was able to learn full, global relations that form digits. This showcases another benefit of the multi-scale model. In contrast to VAEs, the outputs are sharp as normalizing flows can naturally model complex, multi-modal distributions while VAEs have the independent decoder output noise. Nevertheless, the samples from this flow are far from perfect as not all samples show true digits.

Interpolation in latent space

Another popular test for the smoothness of the latent space of generative models is to interpolate between two training examples. As normalizing flows are strictly invertible, we can guarantee that any image is represented in the latent space. We again compare the variational dequantization model with the multi-scale model below.

```
[30]: def interpolate(model, rng, img1, img2, num_steps=8):
    """
    Inputs:
        model - object of ImageFlow class that represents the (trained) flow model
        img1, img2 - Image tensors of shape [1, 28, 28]. Images between which should be
        ↪interpolated.
        num_steps - Number of interpolation steps. 8 interpolation steps mean 6
        ↪intermediate pictures besides img1 and img2
    """
    imgs = np.stack([img1, img2], axis=0)
    z, _, rng = model.encode(imgs, rng)
```

(continues on next page)

(continued from previous page)

```

alpha = jnp.linspace(0, 1, num=num_steps).reshape(-1, 1, 1, 1)
interpolations = z[0:1] * alpha + z[1:2] * (1 - alpha)
interp_imgs, _ = model.sample(interpolations.shape[:1] + imgs.shape[1:], rng=rng, z_
init=interpolations)
show_imgs(interp_imgs, row_size=8)

exmp_imgs, _ = next(iter(train_exmp_loader))

```

[31]:

```

sample_rng = random.PRNGKey(42)
for i in range(2):
    interpolate(flow_dict["vardeq"]["model"].model_bd, sample_rng, exmp_imgs[2*i], exmp_



```

[32]:

```

sample_rng = random.PRNGKey(42)
for i in range(2):
    interpolate(flow_dict["multiscale"]["model"].model_bd, sample_rng, exmp_imgs[2*i],_



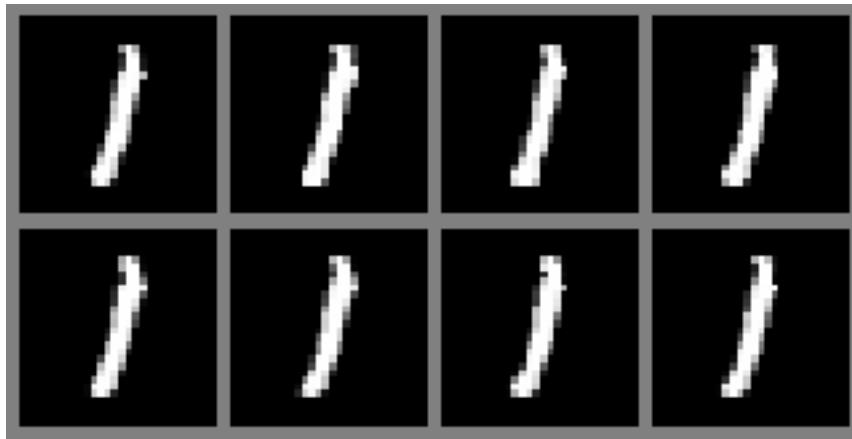
```

The interpolations of the multi-scale model result in more realistic digits (first row 7 \leftrightarrow 8 \leftrightarrow 6, second row 9 \leftrightarrow 6), while the variational dequantization model focuses on local patterns that globally do not form a digit. For the multi-scale model, we actually did not do the “true” interpolation between the two images as we did not consider the variables that were split along the flow (they have been sampled randomly for all samples). However, as we will see in the next experiment, the early variables do not effect the overall image much.

Visualization of latents in different levels of multi-scale

In the following we will focus more on the multi-scale flow. We want to analyse what information is being stored in the variables split at early layers, and what information for the final variables. For this, we sample 8 images where each of them share the same final latent variables, but differ in the other part of the latent variables. Below we visualize three examples of this:

```
[33]: sample_rng = random.PRNGKey(46)
for _ in range(3):
    sample_rng, iter_rng = random.split(sample_rng)
    z_init = random.normal(sample_rng, [1, 7, 7, 8])
    z_init = z_init.repeat(8, 0)
    samples, sample_rng = flow_dict["multiscale"]["model"].model_bd.sample(img_shape=z_
    _init.shape, rng=sample_rng, z_init=z_init)
    show_imgs(samples)
```





We see that the early split variables indeed have a smaller effect on the image. Still, small differences can be spot when we look carefully at the borders of the digits. For instance, the hole at the left of the 0 changes for different samples although all of them represent the same coarse structure. This shows that the flow indeed learns to separate the higher-level information in the final variables, while the early split ones contain local noise patterns.

Visualizing Dequantization

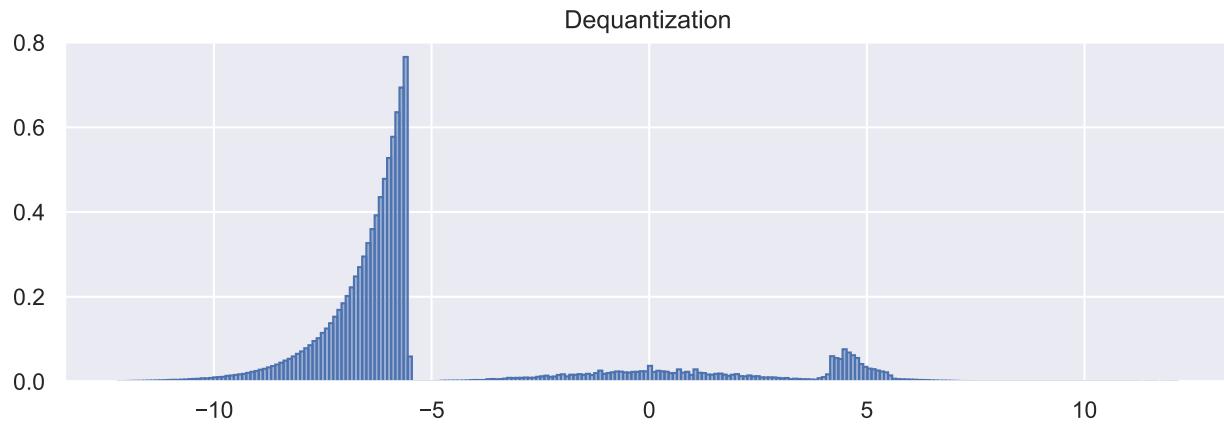
As a final part of this notebook, we will look at the effect of variational dequantization. We have motivated variational dequantization by the issue of sharp edges/boarders being difficult to model, and a flow would rather prefer smooth, prior-like distributions. To check how what noise distribution $q(u|x)$ the flows in the variational dequantization module have learned, we can plot a histogram of output values from the dequantization and variational dequantization module.

```
[34]: def visualize_dequant_distribution(model, rng, imgs, title):
    """
    Inputs:
        model - The flow of which we want to visualize the dequantization distribution
        imgs - Example training images of which we want to visualize the dequantization_
    ↪distribution
    """
    ldj = jnp.zeros(imgs.shape[0], dtype=jnp.float32)
    dequant_vals = []
    for _ in tqdm(range(8), leave=False):
        d, _, rng = model.flows[0](imgs, ldj, rng, reverse=False)
        dequant_vals.append(d)
    dequant_vals = jnp.concatenate(dequant_vals, axis=0)
    dequant_vals = jax.device_get(dequant_vals.reshape(-1))
    sns.set()
    plt.figure(figsize=(10,3))
    plt.hist(dequant_vals, bins=256, color=to_rgb("C0")+(0.5,), edgecolor="C0",_
    ↪density=True)
    if title is not None:
        plt.title(title)
    plt.show()
    plt.close()

sample_imgs, _ = next(iter(train_exmp_loader))
```

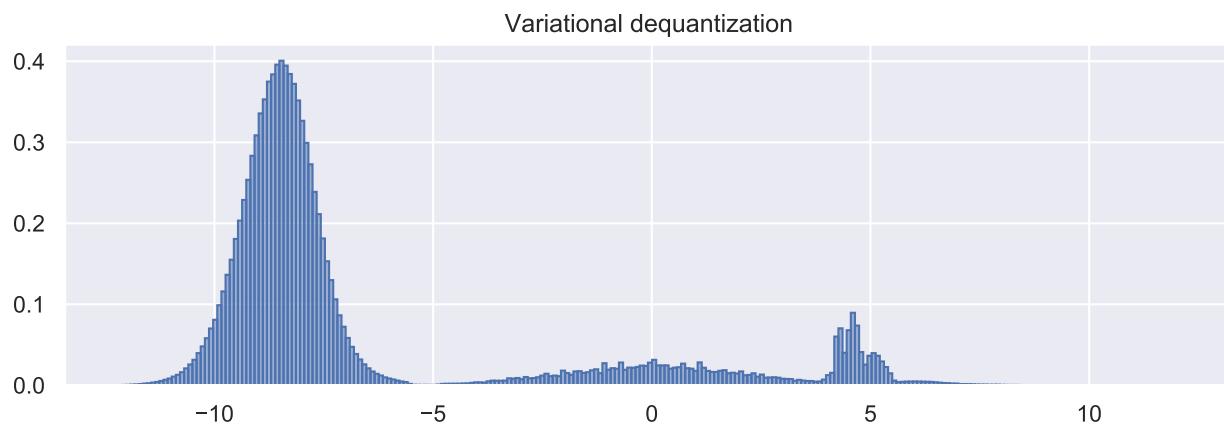
[35]: sample_rng = random.PRNGKey(42)
 visualize_dequant_distribution(flow_dict["simple"]["model"].model_bd, sample_rng, sample_imgs, title="Dequantization")

0% | 0/8 [00:00<?, ?it/s]



[36]: sample_rng = random.PRNGKey(42)
 visualize_dequant_distribution(flow_dict["vardeq"]["model"].model_bd, sample_rng, sample_imgs, title="Variational dequantization")

0% | 0/8 [00:00<?, ?it/s]



The dequantization distribution in the first plot shows that the MNIST images have a strong bias towards 0 (black), and the distribution of them have a sharp border as mentioned before. The variational dequantization module has indeed learned a much smoother distribution with a Gaussian-like curve which can be modeled much better. For the other values, we would need to visualize the distribution $q(u|x)$ on a deeper level, depending on x . However, as all u 's interact and depend on each other, we would need to visualize a distribution in 784 dimensions, which is not that intuitive anymore.

4.26.5 Conclusion

In conclusion, we have seen how to implement our own normalizing flow, and what difficulties arise if we want to apply them on images. Dequantization is a crucial step in mapping the discrete images into continuous space to prevent undesirable delta-peak solutions. While dequantization creates hypercubes with hard border, variational dequantization allows us to fit a flow much better on the data. This allows us to obtain a lower bits per dimension score, while not affecting the sampling speed. The most common flow element, the coupling layer, is simple to implement, and yet effective. Furthermore, multi-scale architectures help to capture the global image context while allowing us to efficiently scale up the flow. Normalizing flows are an interesting alternative to VAEs as they allow an exact likelihood estimate in continuous space, and we have the guarantee that every possible input x has a corresponding latent vector z . However, even beyond continuous inputs and images, flows can be applied and allow us to exploit the data structure in latent space, as e.g. on graphs for the task of molecule generation [6]. Recent advances in [Neural ODEs](#) allow a flow with infinite number of layers, called Continuous Normalizing Flows, whose potential is yet to fully explore. Overall, normalizing flows are an exciting research area which will continue over the next couple of years.

4.26.6 References

- [1] Dinh, L., Sohl-Dickstein, J., and Bengio, S. (2017). “Density estimation using Real NVP,” In: 5th International Conference on Learning Representations, ICLR 2017. [Link](#)
- [2] Kingma, D. P., and Dhariwal, P. (2018). “Glow: Generative Flow with Invertible 1x1 Convolutions,” In: Advances in Neural Information Processing Systems, vol. 31, pp. 10215–10224. [Link](#)
- [3] Ho, J., Chen, X., Srinivas, A., Duan, Y., and Abbeel, P. (2019). “Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design,” in Proceedings of the 36th International Conference on Machine Learning, vol. 97, pp. 2722–2730. [Link](#)
- [4] Durkan, C., Bekasov, A., Murray, I., and Papamakarios, G. (2019). “Neural Spline Flows,” In: Advances in Neural Information Processing Systems, pp. 7509–7520. [Link](#)
- [5] Hoogeboom, E., Cohen, T. S., and Tomczak, J. M. (2020). “Learning Discrete Distributions by Dequantization,” arXiv preprint arXiv2001.11235v1. [Link](#)
- [6] Lippe, P., and Gavves, E. (2021). “Categorical Normalizing Flows via Continuous Transformations,” In: International Conference on Learning Representations, ICLR 2021. [Link](#)

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on [GitHub](#).

4.27 Tutorial 12 (JAX): Autoregressive Image Modeling

Filled notebook:

Pre-trained models:

PyTorch version:

Author: Phillip Lippe

Note: This notebook is written in JAX+Flax. It is a 1-to-1 translation of the original notebook written in PyTorch+PyTorch Lightning with almost identical results. For an introduction to JAX, check out our [Tutorial 2 \(JAX\): Introduction to JAX+Flax](#). Further, throughout the notebook, we comment on major differences to the PyTorch version and provide explanations for the major parts of the JAX code.

Speed comparison: We will report a speed comparison between the PyTorch and JAX/Flax implementation here soon.

Models	PyTorch	JAX
PixelCNN	-min -sec	-min -sec

In this tutorial, we implement an autoregressive likelihood model for the task of image modeling. Autoregressive models are naturally strong generative models that constitute one of the current state-of-the-art architectures on likelihood-based image modeling, and are also the basis for large language generation models such as GPT3. Similar to the language generation you have seen in assignment 2, autoregressive models work on images by modeling the likelihood of a pixel given all previous ones. For instance, in the picture below, we model the pixel x_i as a conditional probability distribution based on all previous (here blue) pixels (figure credit - Aaron van den Oord et al.):

Generally, autoregressive model over high-dimensional data \mathbf{x} factor the joint distribution as the following product of conditionals:

$$p(\mathbf{x}) = p(x_1, \dots, x_n) = \prod_{i=1}^n p(x_i | x_1, \dots, x_{i-1})$$

Learning these conditionals is often much simpler than learning the joint distribution $p(\mathbf{x})$ all together. However, disadvantages of autoregressive models include slow sampling, especially for large images, as we need height-times-width forward passes through the model. In addition, for some applications, we require a latent space as modeled in VAEs and Normalizing Flows. For instance, in autoregressive models, we cannot interpolate between two images because of the lack of a latent representation. We will explore and discuss these benefits and drawbacks alongside with our implementation.

Our implementation will focus on the [PixelCNN](#) [2] model which has been discussed in detail in the lecture. Most current SOTA models use PixelCNN as their fundamental architecture, and various additions have been proposed to improve the performance (e.g. [PixelCNN++](#) and [PixelSNAIL](#)). Hence, implementing PixelCNN is a good starting point for our short tutorial.

First of all, we need to import our standard libraries. Similarly as in the last couple of tutorials, we will use JAX with Flax and Optax here as well.

```
[1]: ## Standard libraries
import os
import math
import numpy as np
from typing import Any

## Imports for plotting
import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
```

(continues on next page)

(continued from previous page)

```

set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import seaborn as sns

## tqdm for progress bars
from tqdm.auto import tqdm

## To run JAX on TPU in Google Colab, uncomment the two lines below
# import jax.tools.colab_tpu
# jax.tools.colab_tpu.setup_tpu()

## JAX
import jax
import jax.numpy as jnp
from jax import random

## Flax (NN in JAX)
try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax
from flax import linen as nn
from flax.training import train_state, checkpoints

## Optax (Optimizers in JAX)
try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax

## PyTorch Data Loading
import torch
import torch.utils.data as data
from torch.utils.tensorboard import SummaryWriter
# Torchvision
from torchvision.datasets import MNIST
from torchvision import transforms

# Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../../saved_models/tutorial12_jax"

print("Device:", jax.devices()[0])
Device: gpu:0

```

We again provide a pretrained model, which is downloaded below:

```
[2]: import urllib.request
from urllib.error import HTTPError
```

(continues on next page)

(continued from previous page)

```
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/JAX/tutorial12/"
# Files to download
pretrained_files = ["PixelCNN.ckpt"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive_"
                  "folder, or contact the author with the full output including the following error:\n", e)
```

Similar to the Normalizing Flows in Tutorial 11, we will work on the MNIST dataset and use 8-bits per pixel (values between 0 and 255). The dataset is loaded below:

```
[3]: # Transformations applied on each image => bring them into a numpy array
# Note that we keep them in the range 0-255 (integers)
def image_to_numpy(img):
    img = np.array(img, dtype=np.int32)
    img = img[..., None] # Make image [28, 28, 1]
    return img

# We need to stack the batch elements
def numpy_collate(batch):
    if isinstance(batch[0], np.ndarray):
        return np.stack(batch)
    elif isinstance(batch[0], (tuple, list)):
        transposed = zip(*batch)
        return [numpy_collate(samples) for samples in transposed]
    else:
        return np.array(batch)

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = MNIST(root=DATASET_PATH, train=True,
                      transform=image_to_numpy, download=True)
train_set, val_set = torch.utils.data.random_split(train_dataset, [50000, 10000],
                                                 generator=torch.Generator().manual_
                                                 .seed(42))
# Loading the test set
test_set = MNIST(root=DATASET_PATH, train=False,
                  transform=image_to_numpy, download=True)

# We define a set of data loaders that we can use for various purposes
train_loader = data.DataLoader(train_set,
```

(continues on next page)

(continued from previous page)

```

batch_size=128,
shuffle=True,
drop_last=True,
collate_fn=numpy_collate,
num_workers=8,
persistent_workers=True)
val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False,
                             drop_last=False, num_workers=4, collate_fn=numpy_collate)
test_loader = data.DataLoader(test_set, batch_size=128, shuffle=False,
                             drop_last=False, num_workers=4, collate_fn=numpy_collate)

```

A good practice is to always visualize some data examples to get an intuition of the data:

```
[4]: def show_imgs(imgs):
    imgs = jax.device_get(imgs)
    if isinstance(imgs, (list, tuple)):
        imgs = np.stack(imgs, axis=0)
    num_imgs = imgs.shape[0]
    if num_imgs < 4 or num_imgs % 4 == 0:
        nrow = min(num_imgs, 4)
        ncol = int(math.ceil(num_imgs/nrow))
    else:
        nrow = num_imgs
        ncol = 1
    imgs = np.pad(imgs, pad_width=[(0,0), (1,1), (1,1), (0,0)], constant_values=128)
    imgs = np.reshape(imgs, (nrow, ncol, *imgs.shape[1:]))
    imgs = np.transpose(imgs, (1, 2, 0, 3, 4))
    imgs = np.reshape(imgs, (imgs.shape[0]*imgs.shape[1], imgs.shape[2]*imgs.shape[3], -1))
    imgs = np.pad(imgs, pad_width=[(1,1), (1,1), (0,0)], constant_values=128)
    imgs = np.squeeze(imgs, axis=-1)
    plt.figure(figsize=(1.5*nrow, 1.5*ncol))
    plt.imshow(imgs, interpolation='nearest', cmap='gray')
    plt.axis('off')
    plt.show()
    plt.close()

show_imgs([train_set[i][0] for i in range(8)])

```



4.27.1 Masked autoregressive convolutions

The core module of PixelCNN is its masked convolutions. In contrast to language models, we don't apply an LSTM on each pixel one-by-one. This would be inefficient because images are grids instead of sequences. Thus, it is better to rely on convolutions that have shown great success in deep CNN classification models.

Nevertheless, we cannot just apply standard convolutions without any changes. Remember that during training of autoregressive models, we want to use teacher forcing which both helps the model training, and significantly reduces the time needed for training. For image modeling, teacher forcing is implemented by using a training image as input to the model, and we want to obtain as output the prediction for each pixel based on *only* its predecessors. Thus, we need to ensure that the prediction for a specific pixel can only be influenced by its predecessors and not by its own value or any “future” pixels. For this, we apply convolutions with a mask.

Which mask we use depends on the ordering of pixels we decide on, i.e. which is the first pixel we predict, which is the second one, etc. The most commonly used ordering is to denote the upper left pixel as the start pixel, and sort the pixels row by row, as shown in the visualization at the top of the tutorial. Thus, the second pixel is on the right of the first one (first row, second column), and once we reach the end of the row, we start in the second row, first column. If we now want to apply this to our convolutions, we need to ensure that the prediction of pixel 1 is not influenced by its own “true” input, and all pixels on its right and in any lower row. In convolutions, this means that we want to set those entries of the weight matrix to zero that take pixels on the right and below into account. As an example for a 5x5 kernel, see a mask below (figure credit - [Aaron van den Oord](#)):

Before looking into the application of masked convolutions in PixelCNN in detail, let's first implement a module that allows us to apply an arbitrary mask to a convolution:

```
[5]: class MaskedConvolution(nn.Module):
    c_out : int
    mask : np.ndarray
    dilation : int = 1

    @nn.compact
    def __call__(self, x):
        # Flax's convolution module already supports masking
        # The mask must be the same size as kernel
        # => extend over input and output feature channels
        if len(self.mask.shape) == 2:
```

(continues on next page)

(continued from previous page)

```

mask_ext = self.mask[...,None,None]
mask_ext = jnp.tile(mask_ext, (1, 1, x.shape[-1], self.c_out))
else:
    mask_ext = self.mask
# Convolution with masking
x = nn.Conv(features=self.c_out,
            kernel_size=self.mask.shape[:2],
            kernel_dilation=self.dilation,
            mask=mask_ext)(x)
return x

```

Vertical and horizontal convolution stacks

To build our own autoregressive image model, we could simply stack a few masked convolutions on top of each other. This was actually the case for the original PixelCNN model, discussed in the paper [Pixel Recurrent Neural Networks](#), but this leads to a considerable issue. When sequentially applying a couple of masked convolutions, the receptive field of a pixel show to have a “blind spot” on the right upper side, as shown in the figure below (figure credit - Aaron van den Oord et al.):

Although a pixel should be able to take into account all other pixels above and left of it, a stack of masked convolutions does not allow us to look to the upper pixels on the right. This is because the features of the pixels above, which we use for convolution, do not contain any information of the pixels on the right of the same row. If they would, we would be “cheating” and actually looking into the future. To overcome this issue, van den Oord et. al [2] proposed to split the convolutions into a vertical and a horizontal stack. The vertical stack looks at all pixels above the current one, while the horizontal takes into account all on the left. While keeping both of them separate, we can actually look at the pixels on the right with the vertical stack without breaking any of our assumptions. The two convolutions are also shown in the figure above.

Let us implement them here as follows:

```
[6]: class VerticalStackConvolution(nn.Module):
    c_out : int
    kernel_size : int
    mask_center : bool = False
    dilation : int = 1

    def setup(self):
        # Mask out all pixels below. For efficiency, we could also reduce the kernel
        # size in height, but for simplicity, we stick with masking here.
        mask = np.ones((self.kernel_size, self.kernel_size), dtype=np.float32)
        mask[self.kernel_size//2+1:,:] = 0
        # For the very first convolution, we will also mask the center row
        if self.mask_center:
            mask[self.kernel_size//2,:] = 0
        # Our convolution module
        self.conv = MaskedConvolution(c_out=self.c_out,
                                      mask=mask,
                                      dilation=self.dilation)

    def __call__(self, x):
```

(continues on next page)

(continued from previous page)

```

    return self.conv(x)

class HorizontalStackConvolution(nn.Module):
    c_out : int
    kernel_size : int
    mask_center : bool = False
    dilation : int = 1

    def setup(self):
        # Mask out all pixels on the left. Note that our kernel has a size of 1
        # in height because we only look at the pixel in the same row.
        mask = np.ones((1, self.kernel_size), dtype=np.float32)
        mask[0, self.kernel_size//2+1:] = 0
        # For the very first convolution, we will also mask the center pixel
        if self.mask_center:
            mask[0, self.kernel_size//2] = 0
        # Our convolution module
        self.conv = MaskedConvolution(c_out=self.c_out,
                                      mask=mask,
                                      dilation=self.dilation)

    def __call__(self, x):
        return self.conv(x)

```

Note that we have an input argument called `mask_center`. Remember that the input to the model is the actual input image. Hence, the very first convolution we apply cannot use the center pixel as input, but must be masked. All consecutive convolutions, however, should use the center pixel as we otherwise lose the features of the previous layer. Hence, the input argument `mask_center` is True for the very first convolutions, and False for all others.

Visualizing the receptive field

To validate our implementation of masked convolutions, we can visualize the receptive field we obtain with such convolutions. We should see that with increasing number of convolutional layers, the receptive field grows in both vertical and horizontal direction, without the issue of a blind spot. The receptive field can be empirically measured by backpropagating an arbitrary loss for the output features of a specific pixel with respect to the input. We implement this idea below, and visualize the receptive field below.

```

[7]: inp_img = np.zeros((1, 11, 11, 1), dtype=np.float32)

def show_center_recep_field(img, apply_fn):
    """
    Calculates the gradients of the input with respect to the output center pixel,
    and visualizes the overall receptive field.

    Inputs:
        img - Input image for which we want to calculate the receptive field on.
        out - Output features/loss which is used for backpropagation, and should be
              the output of the network/computation graph.
    """
    # Determine gradients
    grad_fn = jax.grad(lambda inp: apply_fn(inp)[0,inp.shape[1]//2,inp.shape[2]//2].
    ↪sum()) # L1 loss for simplicity

```

(continues on next page)

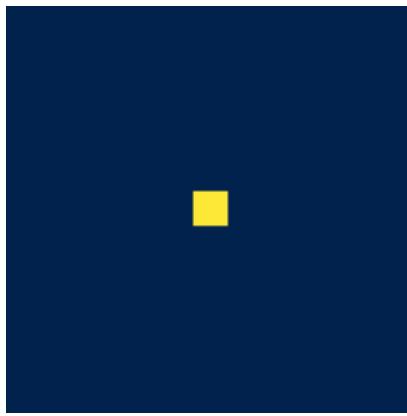
(continued from previous page)

```

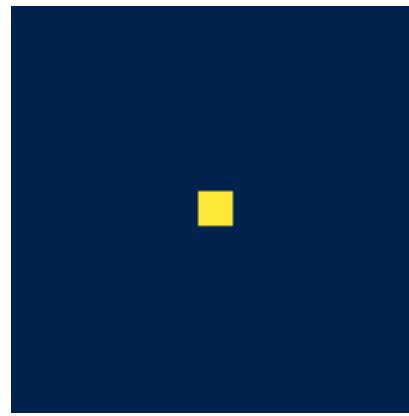


```

Weighted receptive field



Binary receptive field



Let's first visualize the receptive field of a horizontal convolution without the center pixel. We use a small, arbitrary input image (11×11 pixels), and calculate the loss for the center pixel. For simplicity, we initialize all weights with 1 and the bias with 0, and use a single channel. This is sufficient for our visualization purposes.

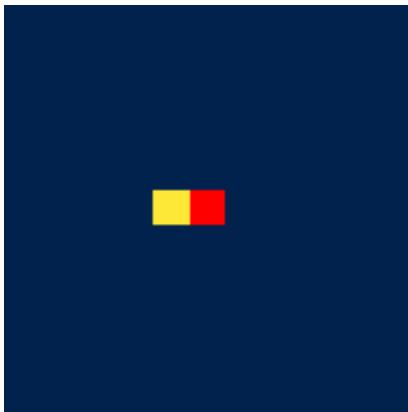
```
[8]: horiz_conv = HorizontalStackConvolution(c_out=1, kernel_size=3, mask_center=True)
# Create parameters with kernel filled with 1, and bias filled with zeros.
# As alternative, one could overwrite kernel init of
init_params = lambda params: jax.tree_map(lambda x: jnp.full(x.shape, (0 if len(x.shape)..
== 1 else 1), dtype=x.dtype), params)
horiz_params = horiz_conv.init(random.PRNGKey(0), inp_img)
```

(continues on next page)

(continued from previous page)

```
horiz_params = init_params(horiz_params)
# Apply horizontal convolution
horiz_conv = horiz_conv.bind(horiz_params)
show_center_recep_field(inp_img, lambda inp: horiz_conv(inp))
```

Weighted receptive field



Binary receptive field

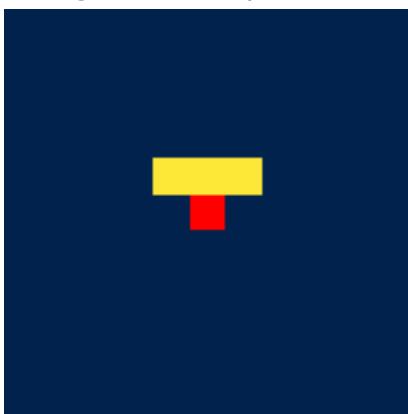


The receptive field is shown in yellow, the center pixel in red, and all other pixels outside of the receptive field are dark blue. As expected, the receptive field of a single horizontal convolution with the center pixel masked and a 3×3 kernel is only the pixel on the left. If we use a larger kernel size, more pixels would be taken into account on the left.

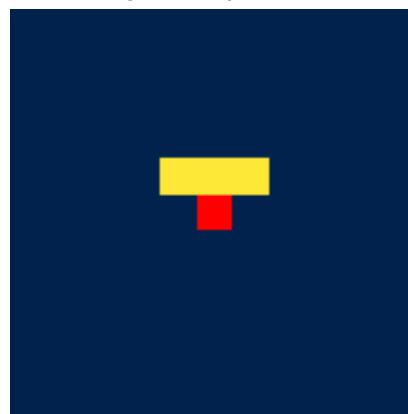
Next, let's take a look at the vertical convolution:

```
[9]: vert_conv = VerticalStackConvolution(c_out=1, kernel_size=3, mask_center=True)
vert_params = vert_conv.init(random.PRNGKey(0), inp_img)
vert_params = init_params(vert_params)
vert_conv = vert_conv.bind(vert_params)
show_center_recep_field(inp_img, lambda inp: vert_conv(inp))
```

Weighted receptive field



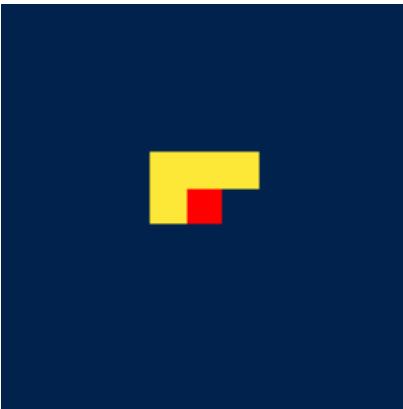
Binary receptive field



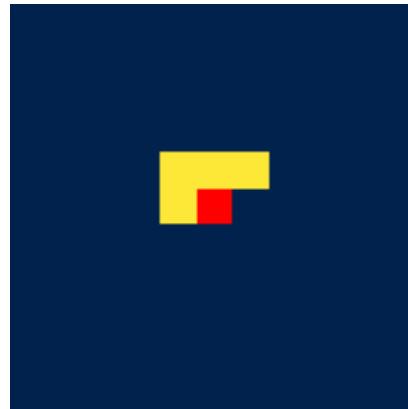
The vertical convolution takes all pixels above into account. Combining these two, we get the L-shaped receptive field of the original masked convolution:

```
[10]: show_center_recep_field(inp_img, lambda inp: vert_conv(inp) + horiz_conv(inp))
```

Weighted receptive field



Binary receptive field



If we stack multiple horizontal and vertical convolutions, we need to take two aspects into account:

1. The center should not be masked anymore for the following convolutions as the features at the pixel's position are already independent of its actual value. If it is hard to imagine why we can do this, just change the value below to `mask_center=True` and see what happens.
2. The vertical convolution is not allowed to work on features from the horizontal convolution. In the feature map of the horizontal convolutions, a pixel contains information about all of the “true” pixels on the left. If we apply a vertical convolution which also uses features from the right, we effectively expand our receptive field to the true input which we want to prevent. Thus, the feature maps can only be merged for the horizontal convolution.

Using this, we can stack the convolutions in the following way. We have two feature streams: one for the vertical stack, and one for the horizontal stack. The horizontal convolutions can operate on the joint features of the previous horizontals and vertical convolutions, while the vertical stack only takes its own previous features as input. For a quick implementation, we can therefore sum the horizontal and vertical output features at each layer, and use those as final output features to calculate the loss on. An implementation of up to 4 consecutive layers is shown below. Note that we reuse the features from the other convolutions with `mask_center=True` from above.

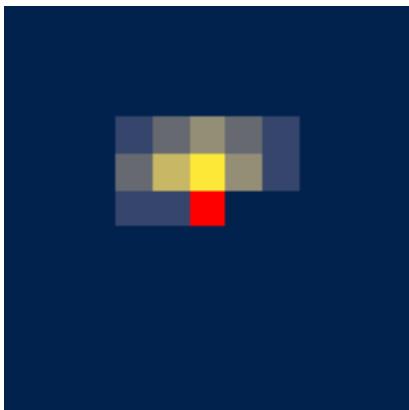
```
[11]: # Convolutions with mask_center=False, no need for new parameters since
horiz_noc_conv = HorizontalStackConvolution(c_out=1, kernel_size=3, mask_center=False)
vert_noc_conv = VerticalStackConvolution(c_out=1, kernel_size=3, mask_center=False)
horiz_noc_conv = horiz_noc_conv.bind(horiz_params)
vert_noc_conv = vert_noc_conv.bind(vert_params)

# We reuse our convolutions for the several layers with same parameters just for visualization.
def num_layer_network(inp, num_layers):
    vert_img = vert_conv(inp)
    horiz_img = horiz_conv(inp) + vert_img
    for _ in range(num_layers-1):
        vert_img = vert_noc_conv(vert_img)
        horiz_img = horiz_noc_conv(horiz_img) + vert_img
    return horiz_img, vert_img

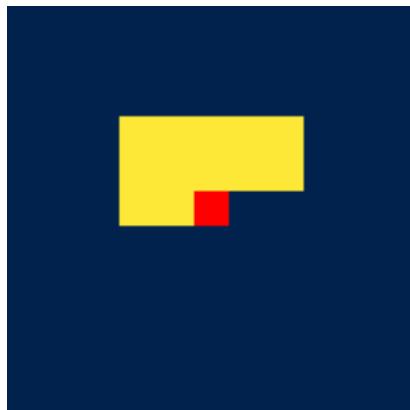
for layer_count in range(2, 6):
    print(f"Layer {layer_count}")
    show_center_recep_field(inp_img, lambda inp: num_layer_network(inp, layer_count)[0])
```

Layer 2

Weighted receptive field

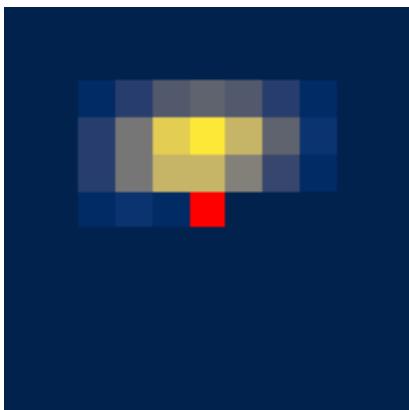


Binary receptive field

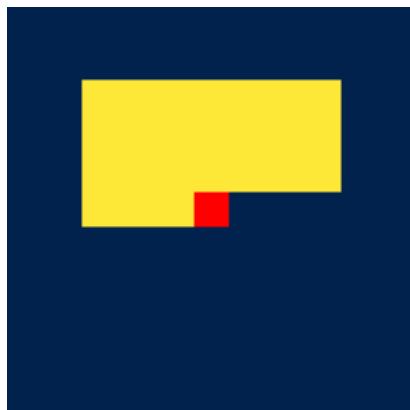


Layer 3

Weighted receptive field

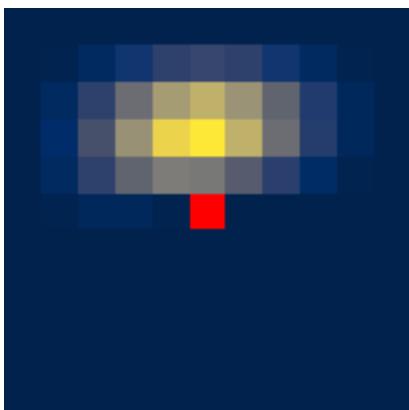


Binary receptive field

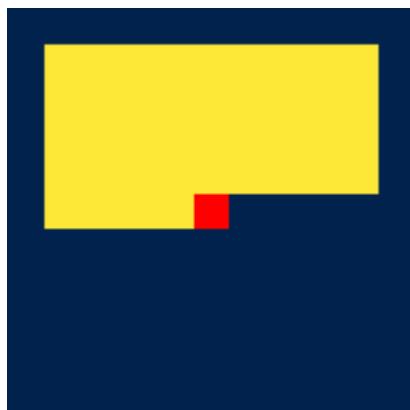


Layer 4

Weighted receptive field

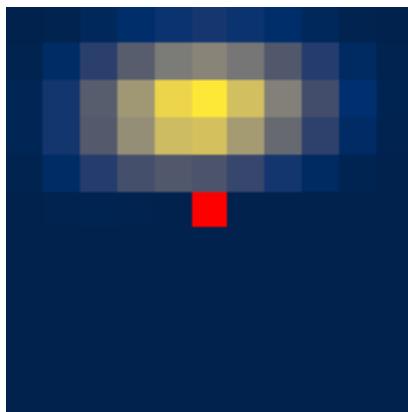


Binary receptive field



Layer 5

Weighted receptive field



Binary receptive field

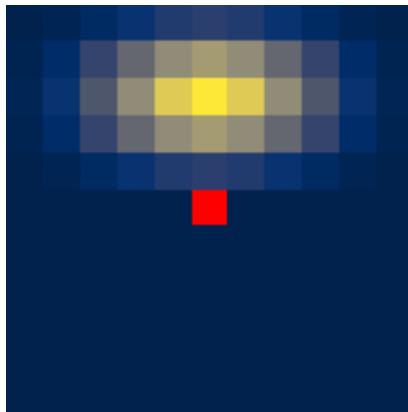


The receptive field above it visualized for the horizontal stack, which includes the features of the vertical convolutions. It grows over layers without any blind spot as we had before. The difference between “weighted” and “binary” receptive field is that for the latter, we check whether there are any gradients flowing back to this pixel. This indicates that the center pixel indeed can use information from this pixel. Nevertheless, due to the convolution weights, some pixels have a stronger effect on the prediction than others. This is visualized in the weighted receptive field by plotting the gradient magnitude for each pixel instead of a binary yes/no.

Another receptive field we can check is the one for the vertical stack as the one above is for the horizontal stack. Let's visualize it below:

```
[12]: show_center_recep_field(inp_img, lambda inp: num_layer_network(inp, 5)[1])
```

Weighted receptive field



Binary receptive field



As we have discussed before, the vertical stack only looks at pixels above the one we want to predict. Hence, we can validate that our implementation works as we initially expected it to.

4.27.2 Gated PixelCNN

In the next step, we will use the masked convolutions to build a full autoregressive model, called Gated PixelCNN. The difference between the original PixelCNN and Gated PixelCNN is the use of separate horizontal and vertical stacks. However, in literature, you often see that people refer to the Gated PixelCNN simply as “PixelCNN”. Hence, in the following, if we say “PixelCNN”, we usually mean the gated version. What “Gated” refers to in the model name is explained next.

Gated Convolutions

For visualizing the receptive field, we assumed a very simplified stack of vertical and horizontal convolutions. Obviously, there are more sophisticated ways of doing it, and PixelCNN uses gated convolutions for this. Specifically, the Gated Convolution block in PixelCNN looks as follows (figure credit - [Aaron van den Oord et al.](#)):

The left path is the vertical stack (the $N \times N$ convolution is masked correspondingly), and the right path is the horizontal stack. Gated convolutions are implemented by having a twice as large output channel size, and combine them by a element-wise multiplication of tanh and a sigmoid. For a linear layer, we can express a gated activation unit as follows:

$$\mathbf{y} = \tanh(\mathbf{W}_f \mathbf{x}) \odot \sigma(\mathbf{W}_g \mathbf{x})$$

For simplicity, biases have been neglected and the linear layer split into two part, \mathbf{W}_f and \mathbf{W}_g . This concept resembles the input and modulation gate in an LSTM, and has been used in many other architectures as well. The main motivation behind this gated activation is that it might allow to model more complex interactions and simplifies learning. But as in any other architecture, this is mostly a design choice and can be considered a hyperparameters.

Besides the gated convolutions, we also see that the horizontal stack uses a residual connection while the vertical stack does not. This is because we use the output of the horizontal stack for prediction. Each convolution in the vertical stack also receives a strong gradient signal as it is only two 1×1 convolutions away from the residual connection, and does not require another residual connection to all its earlier layers.

The implementation in Flax is fairly straight forward for this block, because the visualization above gives us a computation graph to follow:

```
[13]: class GatedMaskedConv(nn.Module):
    dilation : int = 1

    @nn.compact
    def __call__(self, v_stack, h_stack):
        c_in = v_stack.shape[-1]

        # Layers (depend on input shape)
        conv_vert = VerticalStackConvolution(c_out=2*c_in,
                                              kernel_size=3,
                                              mask_center=False,
                                              dilation=self.dilation)
        conv_horiz = HorizontalStackConvolution(c_out=2*c_in,
                                                kernel_size=3,
                                                mask_center=False,
                                                dilation=self.dilation)
        conv_vert_to_horiz = nn.Conv(2*c_in,
                                    kernel_size=(1, 1))
        conv_horiz_1x1 = nn.Conv(c_in,
```

(continues on next page)

(continued from previous page)

```

        kernel_size=(1, 1))

    # Vertical stack (left)
    v_stack_feat = conv_vert(v_stack)
    v_val, v_gate = v_stack_feat.split(2, axis=-1)
    v_stack_out = nn.tanh(v_val) * nn.sigmoid(v_gate)

    # Horizontal stack (right)
    h_stack_feat = conv_horiz(h_stack)
    h_stack_feat = h_stack_feat + conv_vert_to_horiz(v_stack_feat)
    h_val, h_gate = h_stack_feat.split(2, axis=-1)
    h_stack_feat = nn.tanh(h_val) * nn.sigmoid(h_gate)
    h_stack_out = conv_horiz_1x1(h_stack_feat)
    h_stack_out = h_stack_out + h_stack

    return v_stack_out, h_stack_out

```

Building the model

Using the gated convolutions, we can now build our PixelCNN model. The architecture consists of multiple stacked GatedMaskedConv blocks, where we add an additional dilation factor to a few convolutions. This is used to increase the receptive field of the model and allows to take a larger context into account during generation. As a reminder, dilation on a convolution works as follows (figure credit - [Vincent Dumoulin and Francesco Visin](#)):

Note that the smaller output size is only because the animation assumes no padding. In our implementation, we will pad the input image correspondingly. Alternatively to dilated convolutions, we could downsample the input and use an encoder-decoder architecture as in PixelCNN++ [3]. This is especially beneficial if we want to build a very deep autoregressive model. Nonetheless, as we seek to train a reasonably small model, dilated convolutions are the more efficient option to use here.

Below, we implement the PixelCNN model as a Flax module. Besides the stack of gated convolutions, we also have the initial horizontal and vertical convolutions which mask the center pixel, and a final 1×1 convolution which maps the output features to class predictions. To determine the likelihood of a batch of images, we first create our initial features using the masked horizontal and vertical input convolution. Next, we forward the features through the stack of gated convolutions. Finally, we take the output features of the horizontal stack, and apply the 1×1 convolution for classification. We use the bits per dimension metric for the likelihood, similarly to [Tutorial 11](#).

```
[14]: class PixelCNN(nn.Module):
    c_in : int
    c_hidden : int

    def setup(self):
        # Initial convolutions skipping the center pixel
        self.conv_vstack = VerticalStackConvolution(self.c_hidden, kernel_size=3, mask_center=True)
        self.conv_hstack = HorizontalStackConvolution(self.c_hidden, kernel_size=3, mask_center=True)
        # Convolution block of PixelCNN. We use dilation instead of downscaling
        self.conv_layers = [
            GatedMaskedConv(),

```

(continues on next page)

(continued from previous page)

```

        GatedMaskedConv(dilation=2),
        GatedMaskedConv(),
        GatedMaskedConv(dilation=4),
        GatedMaskedConv(),
        GatedMaskedConv(dilation=2),
        GatedMaskedConv()
    ]
# Output classification convolution (1x1)
self.conv_out = nn.Conv(self.c_in * 256, kernel_size=(1, 1))

def __call__(self, x):
    # Forward pass with bpd likelihood calculation
    logits = self.pred_logits(x)
    labels = x.astype(jnp.int32)
    nll = optax.softmax_cross_entropy_with_integer_labels(logits, labels)
    bpd = nll.mean() * np.log2(np.exp(1))
    return bpd

def pred_logits(self, x):
    """
    Forward image through model and return logits for each pixel.
    Inputs:
        x - Image tensor with integer values between 0 and 255.
    """
    # Scale input from 0 to 255 back to -1 to 1
    x = (x.astype(jnp.float32) / 255.0) * 2 - 1

    # Initial convolutions
    v_stack = self.conv_vstack(x)
    h_stack = self.conv_hstack(x)
    # Gated Convolutions
    for layer in self.conv_layers:
        v_stack, h_stack = layer(v_stack, h_stack)
    # 1x1 classification convolution
    # Apply ELU before 1x1 convolution for non-linearity on residual connection
    out = self.conv_out(nn.elu(h_stack))

    # Output dimensions: [Batch, Height, Width, Channels, Classes]
    out = out.reshape(out.shape[0], out.shape[1], out.shape[2], out.shape[3]//256, ↴
256)
    return out

def sample(self, img_shape, rng, img=None):
    """
    Sampling function for the autoregressive model.
    Inputs:
        img_shape - Shape of the image to generate (B,C,H,W)
        img (optional) - If given, this tensor will be used as
                        a starting image. The pixels to fill
                        should be -1 in the input tensor.
    """
    # Create empty image

```

(continues on next page)

(continued from previous page)

```

if img is None:
    img = jnp.zeros(img_shape, dtype=jnp.int32) - 1
    # We jit a prediction step. One could jit the whole loop, but this
    # is expensive to compile and only worth for a lot of sampling calls.
    get_logits = jax.jit(lambda inp: self.pred_logits(inp))
    # Generation loop
    for h in tqdm(range(img_shape[1])), leave=False):
        for w in range(img_shape[2]):
            for c in range(img_shape[3]):
                # Skip if not to be filled (-1)
                if (img[:,h,w,c] != -1).all().item():
                    continue
                # For efficiency, we only have to input the upper part of the image
                # as all other parts will be skipped by the masked convolutions.
anyways
    logits = get_logits(img)
    logits = logots[:,h,w,c,:]
    rng, pix_rng = random.split(rng)
    img = img.at[:,h,w,c].set(random.categorical(pix_rng, logots, axis=-
→1))
return img

```

To sample from the autoregressive model, we need to iterate over all dimensions of the input. We start with an empty image, and fill the pixels one by one, starting from the upper left corner. Note that as for predicting x_i , all pixels below it have no influence on the prediction. Hence, we can cut the image in height without changing the prediction while increasing efficiency. Nevertheless, all the loops in the sampling function already show that it will take us quite some time to sample. A lot of computation could be reused across loop iterations as those the features on the already predicted pixels will not change over iterations. Nevertheless, this takes quite some effort to implement, and is often not done in implementations because in the end, autoregressive sampling remains sequential and slow. Hence, we settle with the default implementation here.

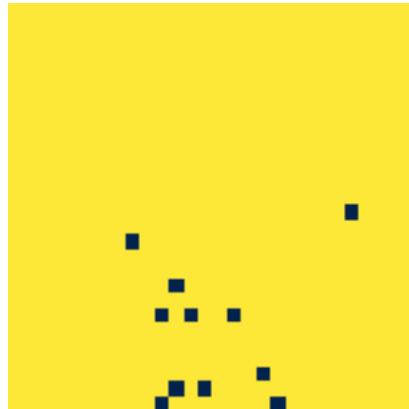
Before training the model, we can check the full receptive field of the model on an MNIST image of size 28×28 :

```
[15]: model = PixelCNN(c_in=1, c_hidden=64)
# inp = random.randint(random.PRNGKey(1), (1,28,28,1), 0, 255).astype(jnp.float32)
inp = jnp.zeros((1,28,28,1))
params = model.init(random.PRNGKey(0), inp)
show_center_recep_field(inp, lambda x: model.bind(params).pred_logits(x)[...,0,0])
```

Weighted receptive field



Binary receptive field



Wait, the receptive field for a single pixel is suddenly the whole image?! Did we make a mistake in our implementation? Unlikely. As it turns out, masked convolutions in Flax show a behavior that leaks gradients through the input mask for larger channel numbers. Note that these gradients for the masked elements is often smaller by a factor of 10^{-6} . What about the gradients for the parameters? Let's check:

```
[16]: grads = jax.grad(lambda p: model.apply(p, inp))(params)
# Gradients of the weights in the first h-stack convolution
# for the first feature map:
grads['params']['conv_hstack']['conv']['Conv_0']['kernel'][..., :, :, 0]

[16]: DeviceArray([[0.10623012],
                 [0.        ],
                 [0.        ]]], dtype=float32)
```

The last two entries indicate the gradients for the center and right pixel of the initial horizontal stack convolution. The gradients are zero, as we expected. So, the gradients for the weights are correct, only the gradients propagating through the feature layers has minor noise that we unintended. Nonetheless, this should be good enough for training.

Note: If you know the details behind why the gradients leak through the mask, I would appreciate any hint on the corresponding [GitHub discussion](#) on Flax.

Despite a large receptive field, keep in mind that this is the “theoretical” receptive field and not necessarily the [effective receptive field](#), which is usually much smaller. For a stronger model, we should therefore try to increase the receptive field even further. Especially, for the pixel on the bottom right, the very last pixel, we would be allowed to take into account the whole image. However, our current receptive field only spans across 1/4 of the image. An encoder-decoder architecture can help with this, but it also shows that we require a much deeper, more complex network in autoregressive models than in VAEs or energy-based models.

Training loop

To train the model, we again can rely on building ourselves a small trainer module that gives us a full training loop, including loading and saving the model.

```
[17]: class TrainerModule:

    def __init__(self,
                 c_in : int,
                 c_hidden : int,
                 exmp_imgs : Any,
                 lr : float = 1e-3,
                 seed : int = 42):
        """
        Module for summarizing all training functionalities for the PixelCNN.
        """
        super().__init__()
        self.lr = lr
        self.seed = seed
        self.model_name = 'PixelCNN'
        # Create empty model. Note: no parameters yet
        self.model = PixelCNN(c_in=c_in, c_hidden=c_hidden)
        # Prepare logging
        self.log_dir = os.path.join(CHECKPOINT_PATH, self.model_name)
        self.logger = SummaryWriter(log_dir=self.log_dir)
        # Create jitted training and eval functions
        self.create_functions()
        # Initialize model
        self.init_model(exmp_imgs)

    def create_functions(self):
        # Training function
        def train_step(state, batch):
            imgs, _ = batch
            loss_fn = lambda params: state.apply_fn(params, imgs)
            loss, grads = jax.value_and_grad(loss_fn)(state.params)
            state = state.apply_gradients(grads=grads)
            return state, loss
        # Eval function
        def eval_step(state, batch):
            imgs, _ = batch
            loss = state.apply_fn(state.params, imgs)
            return loss
        # jit for efficiency
        self.train_step = jax.jit(train_step)
        self.eval_step = jax.jit(eval_step)

    def init_model(self, exmp_imgs):
        # Initialize model
        init_rng = random.PRNGKey(self.seed)
        params = self.model.init(init_rng, exmp_imgs)
        self.state = train_state.TrainState(step=0,
                                            apply_fn=self.model.apply,
```

(continues on next page)

(continued from previous page)

```

        params=params,
        tx=None,
        opt_state=None)

def init_optimizer(self, num_epochs, num_steps_per_epoch):
    # Initialize learning rate schedule and optimizer
    lr_schedule = optax.exponential_decay(
        init_value=self.lr,
        transition_steps=num_steps_per_epoch,
        decay_rate=0.99
    )
    optimizer = optax.adam(lr_schedule)
    # Initialize training state
    self.state = train_state.TrainState.create(apply_fn=self.state.apply_fn,
                                                params=self.state.params,
                                                tx=optimizer)

def train_model(self, train_loader, val_loader, num_epochs=200):
    # Train model for defined number of epochs
    # We first need to create optimizer and the scheduler for the given number of
    epochs
    self.init_optimizer(num_epochs, len(train_loader))
    # Track best eval bpd score.
    best_eval = 1e6
    for epoch_idx in tqdm(range(1, num_epochs+1)):
        self.train_epoch(train_loader, epoch=epoch_idx)
        if epoch_idx % 1 == 0:
            eval_bpd = self.eval_model(val_loader)
            self.logger.add_scalar('val/bpd', eval_bpd, global_step=epoch_idx)
            if eval_bpd <= best_eval:
                best_eval = eval_bpd
                self.save_model(step=epoch_idx)
        self.logger.flush()

def train_epoch(self, train_loader, epoch):
    # Train model for one epoch, and log avg bpd
    avg_loss = 0
    for batch in tqdm(train_loader, desc='Training', leave=False):
        self.state, loss = self.train_step(self.state, batch)
        avg_loss += loss
    avg_loss /= len(train_loader)
    self.logger.add_scalar('train/bpd', avg_loss.item(), global_step=epoch)

def eval_model(self, data_loader):
    # Test model on all images of a data loader and return avg bpd
    avg_bpd, count = 0, 0
    for batch in data_loader:
        bpd = self.eval_step(self.state, batch)
        avg_bpd += bpd * batch[0].shape[0]
        count += batch[0].shape[0]
    eval_bpd = (avg_bpd / count).item()
    return eval_bpd

```

(continues on next page)

(continued from previous page)

```

def save_model(self, step=0):
    # Save current model at certain training iteration
    checkpoints.save_checkpoint(ckpt_dir=self.log_dir,
                                target=self.state.params,
                                step=step,
                                overwrite=True)

def load_model(self, pretrained=False):
    # Load model. We use different checkpoint for pretrained models
    if not pretrained:
        state_dict = checkpoints.restore_checkpoint(ckpt_dir=self.log_dir,
                                                target=None)
    else:
        state_dict = checkpoints.restore_checkpoint(ckpt_dir=os.path.join(CHECKPOINT_PATH,
                                                                f'{self.model_name}.ckpt'), target=None)
        self.state = train_state.TrainState.create(apply_fn=self.state.apply_fn,
                                                params=state_dict,
                                                tx=self.state.tx if self.state.tx is not None else None)
    else optax.sgd(0.1)  # Default optimizer
)

def checkpoint_exists(self):
    # Check whether a pretrained model exist for this autoencoder
    return os.path.isfile(os.path.join(CHECKPOINT_PATH, f'{self.model_name}.ckpt'))

```

Finally, let's write a training function that loads the pretrained model if it exists.

```
[18]: def train_model(max_epochs=150, **model_args):
    # Create a trainer module with specified hyperparameters
    trainer = TrainerModule(exmp_imgs=next(iter(train_loader))[0],
                            **model_args)
    if not trainer.checkpoint_exists():  # Skip training if pretrained model exists
        trainer.train_model(train_loader, val_loader, num_epochs=max_epochs)
        trainer.load_model()
    else:
        trainer.load_model(pretrained=True)
    val_bpd = trainer.eval_model(val_loader)
    test_bpd = trainer.eval_model(test_loader)
    # Bind parameters to model for easier inference
    trainer.model_bd = trainer.model.bind(trainer.state.params)
    return trainer, {'val_bpd': val_bpd, 'test_bpd': test_bpd}
```

Training the model is time consuming and we recommend using the provided pre-trained model for going through this notebook. However, feel free to play around with the hyperparameter like number of layers etc. if you want to get a feeling for those.

When calling the training function with a pre-trained model, we automatically load it and print its test performance:

```
[19]: trainer, result = train_model(max_epochs=150, c_in=1, c_hidden=64)
print(f'Test bpd: {result["test_bpd"]:.4f}')

Test bpd: 0.820
```

With a test performance of 0.820 bpd, the PixelCNN significantly outperforms the normalizing flows we have seen in [Tutorial 11](#). Considering image modeling as an autoregressive problem simplifies the learning process as predicting one pixel given the ground truth of all others is much easier than predicting all pixels at once. In addition, PixelCNN can explicitly predict the pixel values by a discrete softmax while Normalizing Flows have to learn transformations in continuous latent space. These two aspects allow the PixelCNN to achieve a notably better performance.

To fully compare the models, let's also measure the number of parameters of the PixelCNN:

```
[20]: num_params = sum([np.prod(p.shape) for p in jax.tree_leaves(trainer.state.params)])
print("Number of parameters: {}".format(num_params))
```

Number of parameters: 852,160

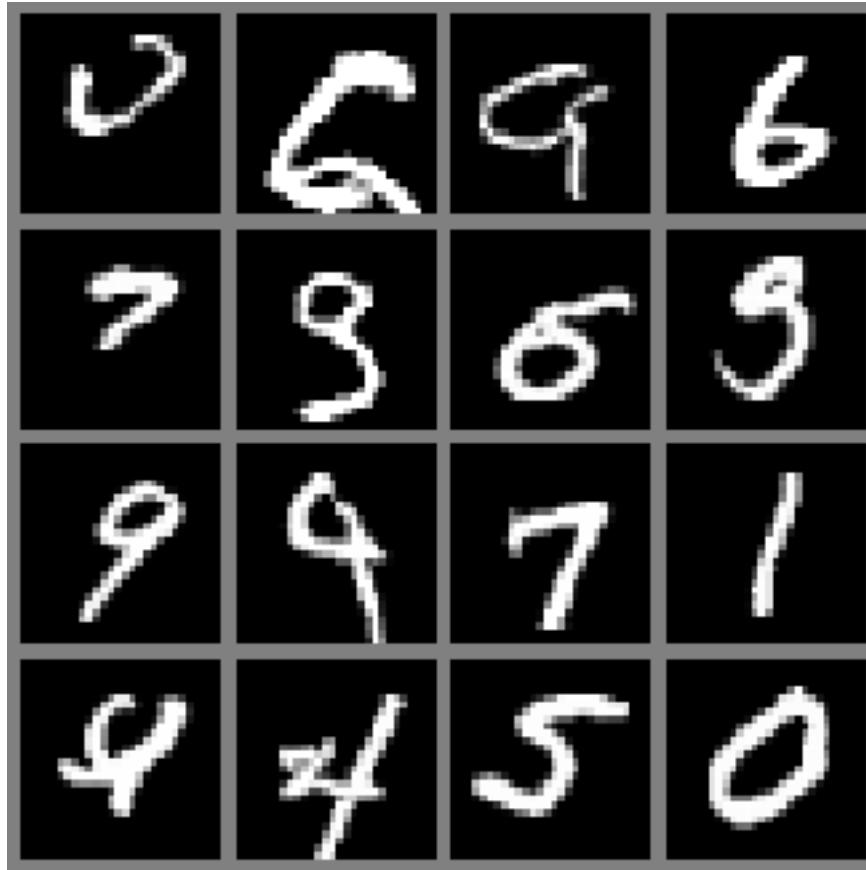
Compared to the multi-scale normalizing flows, the PixelCNN has considerably less parameters. Of course, the number of parameters depend on our hyperparameter choices. Nevertheless, in general, it can be said that autoregressive models require considerably less parameters than normalizing flows to reach good performance, based on the reasons stated above. Still, autoregressive models are much slower in sampling than normalizing flows, which limits their possible applications.

4.27.3 Sampling

One way of qualitatively analysing generative models is by looking at the actual samples. Let's therefore use our sampling function to generate a few digits:

```
[21]: samples = trainer.model_bd.sample(img_shape=(16, 28, 28, 1), rng=random.PRNGKey(0))
show_imgs(samples)
```

0% | 0/28 [00:00<?, ?it/s]



Most of the samples can be identified as digits, and overall we achieve a better quality than we had in normalizing flows. This goes along with the lower likelihood we achieved with autoregressive models. Nevertheless, we also see that there is still place for improvement as a considerable amount of samples cannot be identified (for example the first row). Deeper autoregressive models are expected to achieve better quality, as they can take more context into account for generating the pixels.

The trained model itself is not restricted to any specific image size. However, what happens if we actually sample a larger image than we had seen in our training dataset? Let's try below to sample images of size 64×64 instead of 28×28 :

```
[22]: samples = trainer.model_bd.sample(img_shape=(8, 64, 64, 1), rng=random.PRNGKey(0))
show_imgs(samples)
```

0% | 0/64 [00:00<?, ?it/s]



The larger images show that changing the size of the image during testing confuses the model and generates abstract figures (you can sometimes spot a digit in the upper left corner). In addition, sampling for images of 64x64 pixels take up to a minute on a GPU. Clearly, autoregressive models cannot be scaled to large images without changing the sampling procedure such as with [forecasting](#). Our implementation is also not the most efficient as many computations can be stored and reused throughout the sampling process. Nevertheless, the sampling procedure stays sequential which is inherently slower than parallel generation like done in normalizing flows.

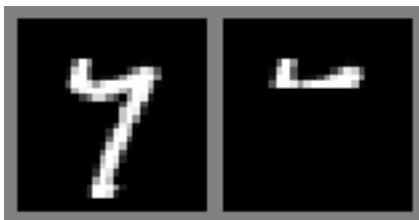
Autocompletion

One common application done with autoregressive models is auto-completing an image. As autoregressive models predict pixels one by one, we can set the first N pixels to predefined values and check how the model completes the image. For implementing this, we just need to skip the iterations in the sampling loop that already have a value unequal -1. See above in our Flax module for the specific implementation. In the cell below, we randomly take three images from the training set, mask about the lower half of the image, and let the model autocomplete it. To see the diversity of samples, we do this 12 times for each image:

```
[23]: def autocomplete_image(img):
    # Remove lower half of the image
    img_init = np.copy(img)
    img_init[10:] = -1
    print("Original image and input image to sampling:")
    show_imgs([img, img_init])
    # Generate 12 example completions
    img_init = np.repeat(img_init[None], 12, axis=0)
    img_init = jax.device_put(img_init)
    img_generated = trainer.model_bd.sample(img_init.shape,
                                             rng=random.PRNGKey(42),
                                             img=img_init)
    print("Autocompletion samples:")
    show_imgs(img_generated)

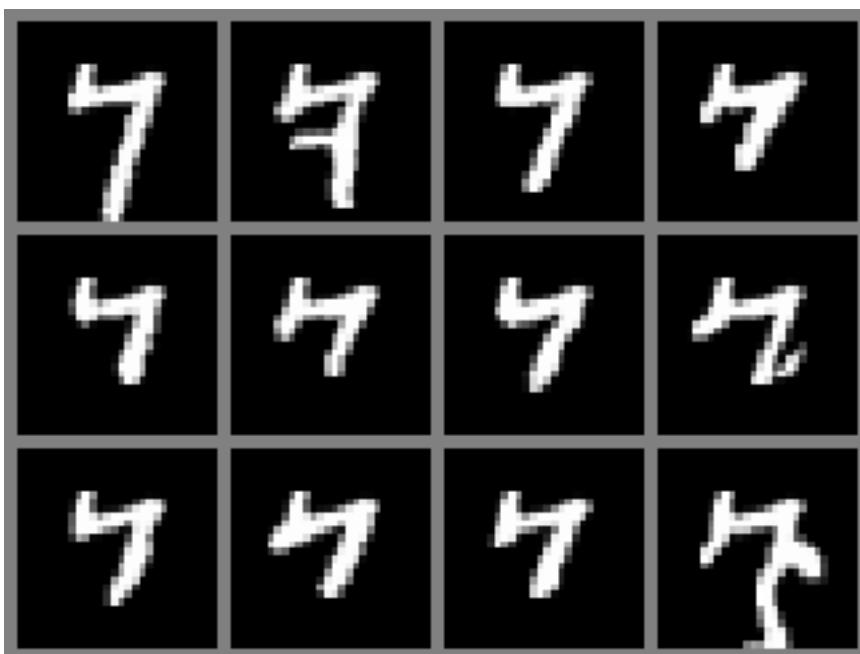
for i in range(1,4):
    img = train_set[i][0]
    autocomplete_image(img)
```

Original image and input image to sampling:



0% | 0/28 [00:00<?, ?it/s]

Autocompletion samples:

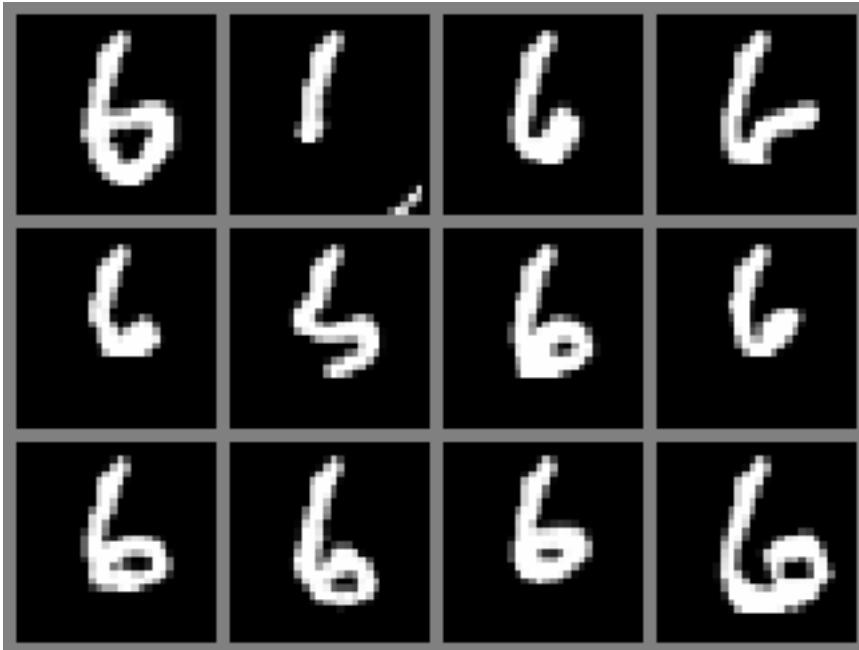


Original image and input image to sampling:



0% | 0/28 [00:00<?, ?it/s]

Autocompletion samples:

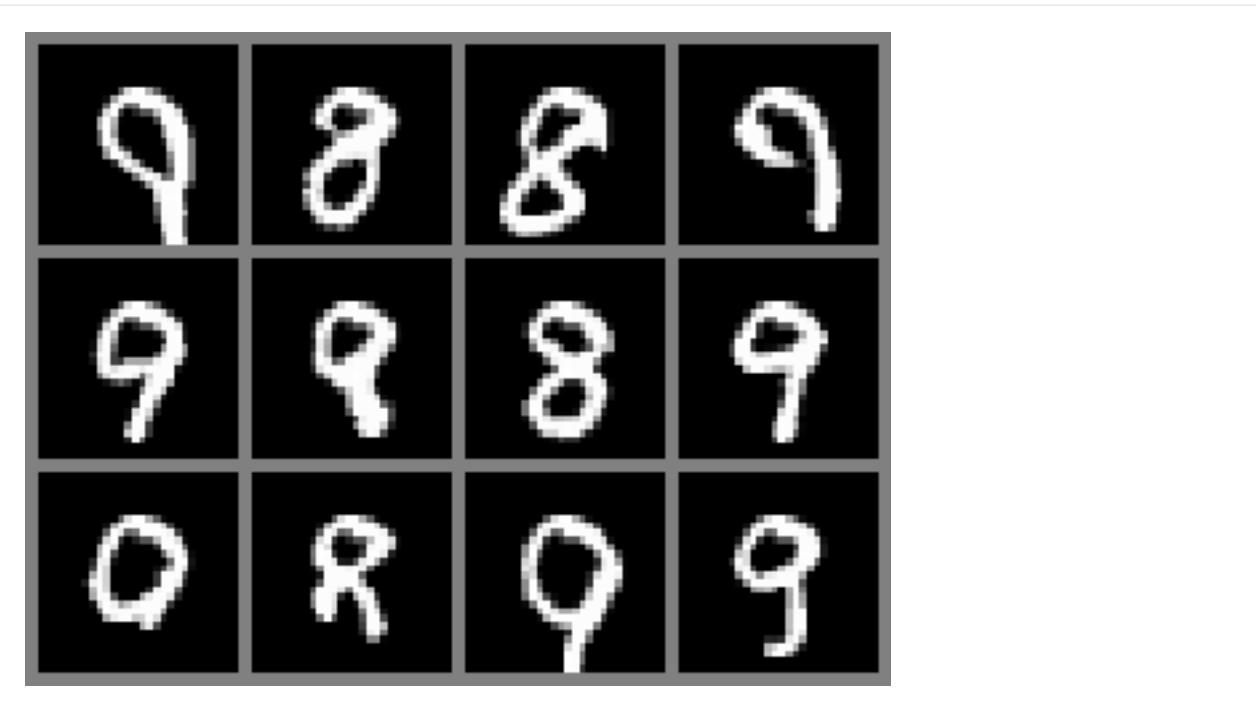


Original image and input image to sampling:



0% | 0/28 [00:00<?, ?it/s]

Autocompletion samples:



For the first two digits (7 and 6), we see that the 12 samples all result in a shape which resemble the original digit. Nevertheless, there are some style difference in writing the 7, and some deformed sixes in the samples. When auto-completing the 9 below, we see that the model can fit multiple digits to it. We obtain diverse samples from 0, 8 and 9. This shows that despite having no latent space, we can still obtain diverse samples from an autoregressive model.

Visualization of the predictive distribution (softmax)

Autoregressive models use a softmax over 256 values to predict the next pixel. This gives the model a large flexibility as the probabilities for each pixel value can be learned independently if necessary. However, the values are actually not independent because the values 32 and 33 are much closer than 32 and 255. In the following, we visualize the softmax distribution that the model predicts to gain insights how it has learned the relationships of close-by pixels.

To do this, we first run the model on a batch of images and store the output softmax distributions:

```
[24]: det_loader = data.DataLoader(train_set, batch_size=128, shuffle=False, drop_last=False,
    collate_fn=numpy_collate)
imgs, _ = next(iter(det_loader))

out = trainer.model_bd.pred_logits(imgs)
out = nn.softmax(out, axis=-1)
mean_out = jax.device_get(out.mean(axis=[0,1,2,3]))
out = jax.device_get(out)
```

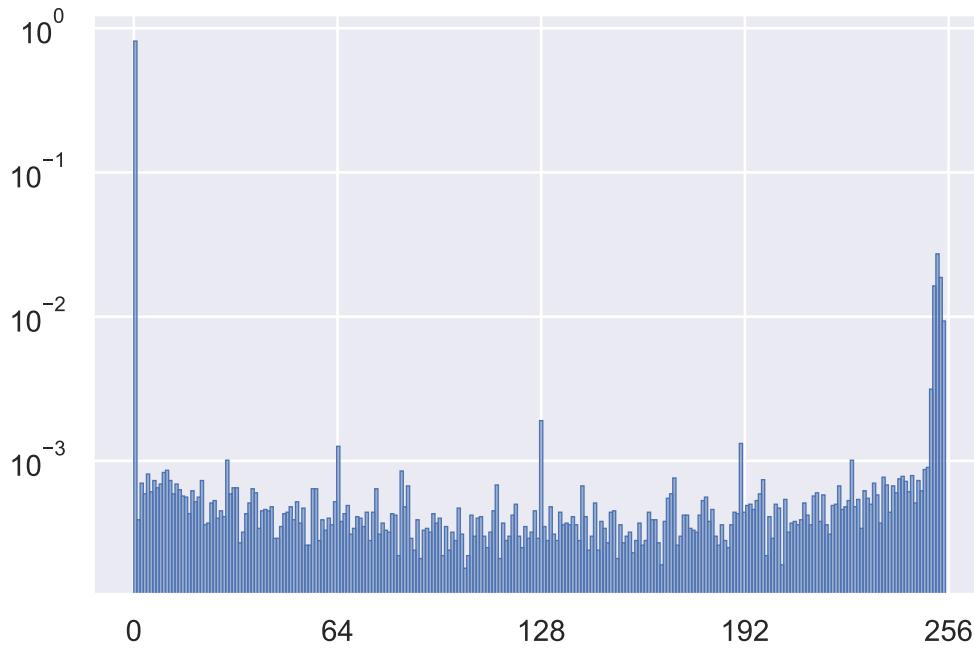
Before diving into the model, let's visualize the distribution of the pixel values in the whole dataset:

```
[25]: sns.set()
plot_args = {"color": to_rgb("C0")+(0.5,), "edgecolor": "C0", "linewidth": 0.5, "width": 1.0}
plt.hist(imgs.reshape(-1), bins=256, density=True, **plot_args)
plt.yscale("log")
```

(continues on next page)

(continued from previous page)

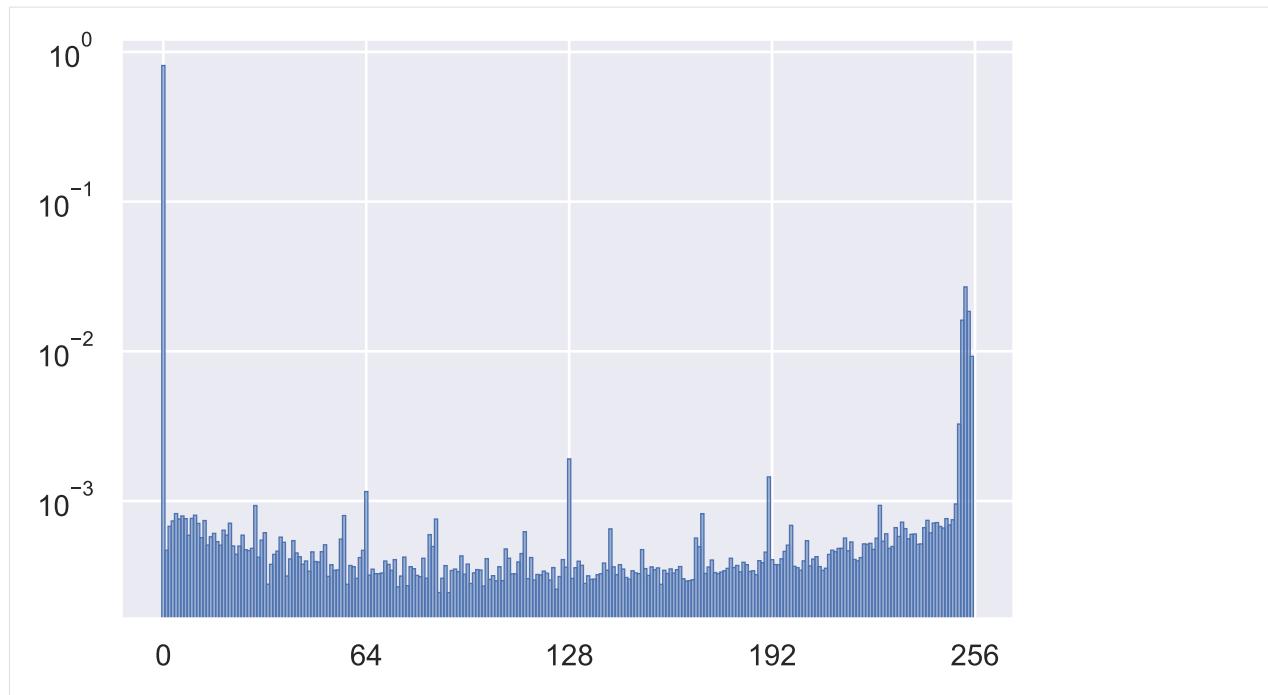
```
plt.xticks([0,64,128,192,256])
plt.show()
plt.close()
```



As we would expect from the seen images, the pixel value 0 (black) is the dominant value, followed by a batch of values between 250 and 255. Note that we use a log scale on the y-axis due to the big imbalance in the dataset. Interestingly, the pixel values 64, 128 and 191 also stand out which is likely due to the quantization used during the creation of the dataset. For RGB images, we would also see two peaks around 0 and 255, but the values in between would be much more frequent than in MNIST (see Figure 1 in the [PixelCNN++](#) for a visualization on CIFAR10).

Next, we can visualize the distribution our model predicts (in average):

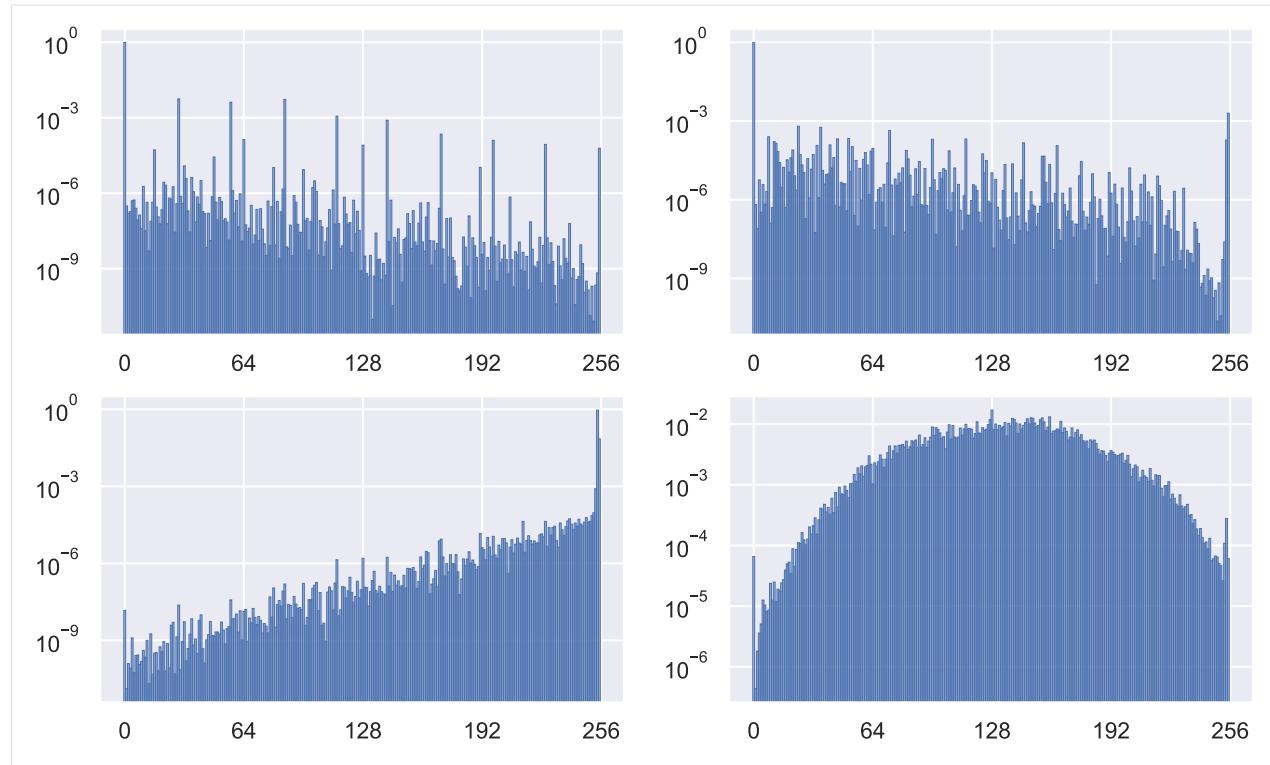
```
[26]: plt.bar(np.arange(mean_out.shape[0]), mean_out, **plot_args)
plt.yscale("log")
plt.xticks([0,64,128,192,256])
plt.show()
plt.close()
```



This distribution is very close to the actual dataset distribution. This is in general a good sign, but we can see a slightly smoother histogram than above.

Finally, to take a closer look at learned value relations, we can visualize the distribution for individual pixel predictions to get a better intuition. For this, we pick 4 random images and pixels, and visualize their distribution below:

```
[27]: fig, ax = plt.subplots(2,2, figsize=(10,6))
for i in range(4):
    ax_sub = ax[i//2][i%2]
    ax_sub.bar(np.arange(out.shape[-1], dtype=np.int32), out[i+4,14,14,0], **plot_args)
    ax_sub.set_yscale("log")
    ax_sub.set_xticks([0,64,128,192,256])
plt.show()
plt.close()
```



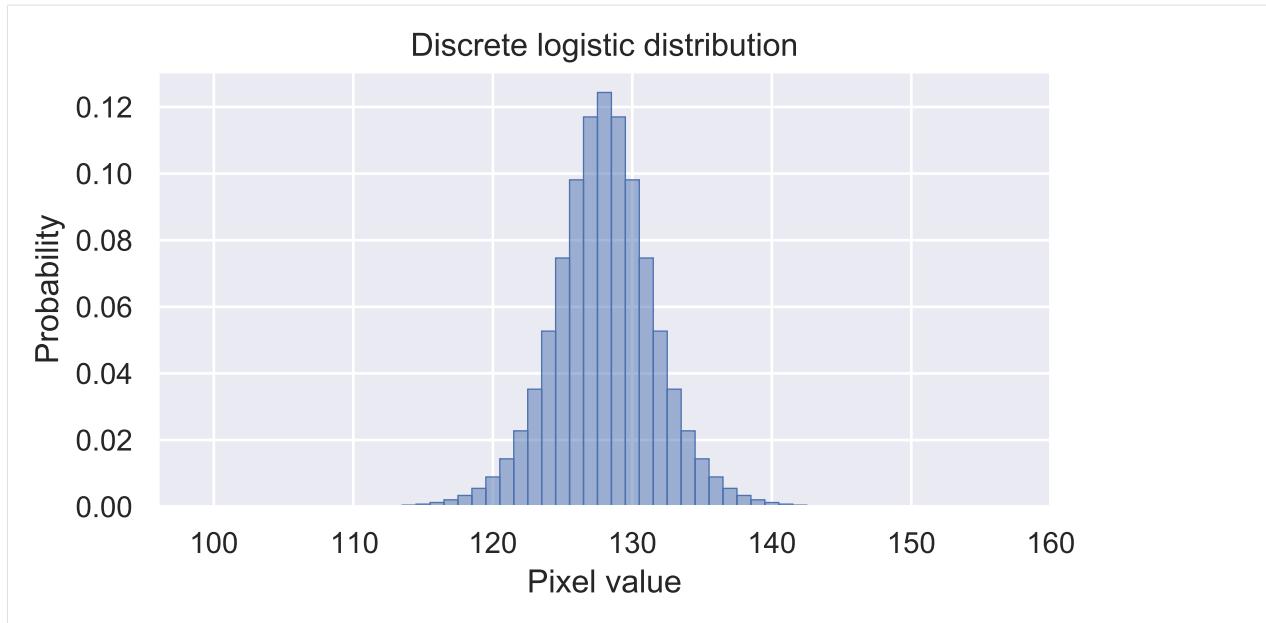
Overall we see a very diverse set of distributions, with a usual peak for 0 and close to 1. However, the distributions in the first row show a potentially undesirable behavior. For instance, the value 242 has a 1000x lower likelihood than 243 although they are extremely close and can often not be distinguished. This shows that the model might have not generalized well over pixel values. The better solution to this problem is to use discrete logistics mixtures instead of a softmax distribution. A discrete logistic distribution can be imagined as discretized, binned Gaussians. Using a mixture of discrete logistics instead of a softmax introduces an inductive bias to the model to assign close-by values similar likelihoods. We can visualize a discrete logistic below:

```
[28]: mu = np.array([128])
sigma = np.array([2.0])

def discrete_logistic(x, mu, sigma):
    return nn.sigmoid((x+0.5-mu)/sigma) - nn.sigmoid((x-0.5-mu)/sigma)

x = np.arange(256)
p = discrete_logistic(x, mu, sigma)

# Visualization
plt.figure(figsize=(6,3))
plt.bar(x, p, **plot_args)
plt.xlim(96,160)
plt.title("Discrete logistic distribution")
plt.xlabel("Pixel value")
plt.ylabel("Probability")
plt.show()
plt.close()
```



Instead of the softmax, the model would output mean and standard deviations for the K logistics we use in the mixture. This is one of the improvements in autoregressive models that PixelCNN++ [3] has introduced compared to the original PixelCNN.

4.27.4 Conclusion

In this tutorial, we have looked at autoregressive image modeling, and implemented the PixelCNN architecture. With the usage of masked convolutions, we are able to apply a convolutional network in which a pixel is only influenced by all its predecessors. Separating the masked convolution into a horizontal and vertical stack allowed us to remove the known blind spot on the right upper row of a pixel. In experiments, autoregressive models outperformed normalizing flows in terms of bits per dimension, but are much slower to sample from. Improvements, that we have not implemented ourselves here, are discrete logistic mixtures, a downsampling architecture, and changing the pixel order in a diagonal fashion (see PixelSNAIL). Overall, autoregressive models are another, strong family of generative models, which however are mostly used in sequence tasks because of their linear scaling in sampling time than quadratic as on images.

4.27.5 References

- [1] van den Oord, A., et al. “Pixel Recurrent Neural Networks.” arXiv preprint arXiv:1601.06759 (2016). [Link](#)
- [2] van den Oord, A., et al. “Conditional Image Generation with PixelCNN Decoders.” In Advances in Neural Information Processing Systems 29, pp. 4790–4798 (2016). [Link](#)
- [3] Salimans, Tim, et al. “PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications.” arXiv preprint arXiv:1701.05517 (2017). [Link](#)

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.28 Tutorial 15 (JAX): Vision Transformers

Filled notebook:

Pre-trained models:

PyTorch version:

Author: Phillip Lippe

Note: This notebook is written in JAX+Flax. It is a 1-to-1 translation of the original notebook written in PyTorch+PyTorch Lightning with almost identical results. For an introduction to JAX, check out our [Tutorial 2 \(JAX\): Introduction to JAX+Flax](#). Further, throughout the notebook, we comment on major differences to the PyTorch version and provide explanations for the major parts of the JAX code.

Speed comparison: We note the training times for all models in the PyTorch and the JAX implementation below (PyTorch v1.11, JAX v0.3.13). The models were trained on the same hardware (NVIDIA RTX3090, 24 core CPU) and we slightly adjusted the tutorials to use the exact same training settings (200 epochs, data loading parameters, evaluation schedule, etc.). Overall, the JAX implementation is just slightly faster (1.1x) than PyTorch.

Models	PyTorch	JAX
Vision Transformer	28min 40sec	27min 10sec

In this tutorial, we will take a closer look at a recent new trend: Transformers for Computer Vision. Since [Alexey Dosovitskiy et al.](#) successfully applied a Transformer on a variety of image recognition benchmarks, there have been an incredible amount of follow-up works showing that CNNs might not be optimal architecture for Computer Vision anymore. But how do Vision Transformers work exactly, and what benefits and drawbacks do they offer in contrast to CNNs? We will answer these questions by implementing a Vision Transformer ourselves and train it on the popular, small dataset CIFAR10. We will compare these results to the convolutional architectures of [Tutorial 5](#).

If you are not familiar with Transformers yet, take a look at [Tutorial 6](#) where we discuss the fundamentals of Multi-Head Attention and Transformers. Let's start with importing our standard set of libraries.

```
[1]: ## Standard libraries
import os
import numpy as np
import math
import json
from functools import partial
from PIL import Image
from collections import defaultdict

## Imports for plotting
```

(continues on next page)

(continued from previous page)

```

import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## tqdm for progress bars
from tqdm.auto import tqdm

## To run JAX on TPU in Google Colab, uncomment the two lines below
# import jax.tools.colab_tpu
# jax.tools.colab_tpu.setup_tpu()

## JAX
import jax
import jax.numpy as jnp
from jax import random

## Flax (NN in JAX)
try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax
from flax import linen as nn
from flax.training import train_state, checkpoints

## Optax (Optimizers in JAX)
try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax

## PyTorch
import torch
import torch.utils.data as data
from torch.utils.tensorboard import SummaryWriter
import torchvision
from torchvision import transforms
from torchvision.datasets import CIFAR10

# Import tensorboard
%load_ext tensorboard

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../..../data"

```

(continues on next page)

(continued from previous page)

```
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = ".../.../saved_models/tutorial15_jax"

# Seeding for random operations
main_rng = random.PRNGKey(42)

print("Device:", jax.devices()[0])

/tmp/ipykernel_2627394/2327484937.py:16: DeprecationWarning: `set_matplotlib_formats` is
  deprecated since IPython 7.23, directly use `matplotlib_inline.backend_inline.set_
  _matplotlib_formats()`
    set_matplotlib_formats('svg', 'pdf') # For export

Device: gpu:0
```

We provide a pre-trained Vision Transformer which we download in the next cell. However, Vision Transformers can be relatively quickly trained on CIFAR10 with an overall training time of less than an hour on an NVIDIA TitanRTX. Feel free to experiment with training your own Transformer once you went through the whole notebook.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/JAX/"
# Files to download
pretrained_files = ["tutorial15/ViT.ckpt", "tutorial15/tensorboards/ViT/events.out.
˓→tfevents.ViT",
                    "tutorial5/tensorboards/ResNet/events.out.tfevents.resnet"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name.split("/", 1)[1])
    if "/" in file_name.split("/", 1)[1]:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive_
  ↵folder, or contact the author with the full output including the following error:\n", e)
```

We load the CIFAR10 dataset below. We use the same setup of the datasets and data augmentations as for the CNNs in Tutorial 5 to keep a fair comparison. The constants in the image normalization correspond to the values that scale and shift the data to a zero mean and standard deviation of one.

```
[3]: # Transformations applied on each image => bring them into a numpy array
DATA_MEANS = np.array([0.49139968, 0.48215841, 0.44653091])
DATA_STD = np.array([0.24703223, 0.24348513, 0.26158784])
def image_to_numpy(img):
```

(continues on next page)

(continued from previous page)

```



```

(continues on next page)

(continued from previous page)

```
test_loader = data.DataLoader(test_set,
                                batch_size=128,
                                shuffle=False,
                                drop_last=False,
                                collate_fn=numpy_collate,
                                num_workers=4,
                                persistent_workers=True)
```

Files already downloaded and verified

Files already downloaded and verified

Files already downloaded and verified

```
[4]: # Visualize some examples
def numpy_to_torch(array):
    array = jax.device_get(array)
    tensor = torch.from_numpy(array)
    tensor = tensor.permute(0, 3, 1, 2)
    return tensor

NUM_IMAGES = 4
CIFAR_images = np.stack([val_set[idx][0] for idx in range(NUM_IMAGES)], axis=0)
img_grid = torchvision.utils.make_grid(numpy_to_torch(CIFAR_images),
                                       nrow=4, normalize=True, pad_value=0.9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("Image examples of the CIFAR10 dataset")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

Image examples of the CIFAR10 dataset



4.28.1 Transformers for image classification

Transformers have been originally proposed to process sets since it is a permutation-equivariant architecture, i.e., producing the same output permuted if the input is permuted. To apply Transformers to sequences, we have simply added a positional encoding to the input feature vectors, and the model learned by itself what to do with it. So, why not do the same thing on images? This is exactly what Alexey Dosovitskiy et al. proposed in their paper “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. Specifically, the Vision Transformer is a model for image classification that views images as sequences of smaller patches. As a preprocessing step, we split an image of, for example, 48×48 pixels into $9 \times 16 \times 16$ patches. Each of those patches is considered to be a “word”/“token” and projected to a feature space. With adding positional encodings and a token for classification on top, we can apply a Transformer as usual to this sequence and start training it for our task. A nice GIF visualization of the architecture is shown below (figure credit - Phil Wang):

We will walk step by step through the Vision Transformer, and implement all parts by ourselves. First, let’s implement the image preprocessing: an image of size $N \times N$ has to be split into $(N/M)^2$ patches of size $M \times M$. These represent the input words to the Transformer.

```
[5]: def img_to_patch(x, patch_size, flatten_channels=True):
    """
    Inputs:
        x - torch.Tensor representing the image of shape [B, H, W, C]
        patch_size - Number of pixels per dimension of the patches (integer)
        flatten_channels - If True, the patches will be returned in a flattened format
                           as a feature vector instead of a image grid.
    """
    B, H, W, C = x.shape
    x = x.reshape(B, H//patch_size, patch_size, W//patch_size, patch_size, C)
    x = x.transpose(0, 1, 3, 2, 4, 5)      # [B, H', W', p_H, p_W, C]
    x = x.reshape(B, -1, *x.shape[3:])    # [B, H'*W', p_H, p_W, C]
    if flatten_channels:
        x = x.reshape(B, x.shape[1], -1) # [B, H'*W', p_H*p_W*C]
    return x
```

Let’s take a look at how that works for our CIFAR examples above. For our images of size 32×32 , we choose a patch size of 4. Hence, we obtain sequences of 64 patches of size 4×4 . We visualize them below:

```
[6]: img_patches = img_to_patch(CIFAR_images, patch_size=4, flatten_channels=False)

fig, ax = plt.subplots(CIFAR_images.shape[0], 1, figsize=(14,3))
fig.suptitle("Images as input sequences of patches")
for i in range(CIFAR_images.shape[0]):
    img_grid = torchvision.utils.make_grid(numpy_to_torch(img_patches[i]),
                                           nrow=64, normalize=True, pad_value=0.9)
    img_grid = img_grid.permute(1, 2, 0)
    ax[i].imshow(img_grid)
    ax[i].axis('off')
plt.show()
plt.close()
```



Compared to the original images, it is much harder to recognize the objects from those patch lists now. Still, this is the input we provide to the Transformer for classifying the images. The model has to learn itself how it has to combine the patches to recognize the objects. The inductive bias in CNNs that an image is a grid of pixels, is lost in this input format.

After we have looked at the preprocessing, we can now start building the Transformer model. Since we have discussed the fundamentals of Multi-Head Attention in [Tutorial 6](#), we will use the Flax module `nn.MultiHeadDotProductAttention` ([docs](#)) here. Further, we use the Pre-Layer Normalization version of the Transformer blocks proposed by [Ruimin Xiong et al.](#) in 2020. The idea is to apply Layer Normalization not in between residual blocks, but instead as a first layer in the residual blocks. This reorganization of the layers supports better gradient flow and removes the necessity of a warm-up stage. A visualization of the difference between the standard Post-LN and the Pre-LN version is shown below.

The implementation of the Pre-LN attention block looks as follows:

```
[7]: class AttentionBlock(nn.Module):
    embed_dim : int    # Dimensionality of input and attention feature vectors
    hidden_dim : int   # Dimensionality of hidden layer in feed-forward network
    num_heads : int    # Number of heads to use in the Multi-Head Attention block
    dropout_prob : float = 0.0 # Amount of dropout to apply in the feed-forward network

    def setup(self):
        self.attn = nn.MultiHeadDotProductAttention(num_heads=self.num_heads)
        self.linear = [
            nn.Dense(self.hidden_dim),
            nn.gelu,
            nn.Dropout(self.dropout_prob),
            nn.Dense(self.embed_dim)
        ]
        self.layer_norm_1 = nn.LayerNorm()
        self.layer_norm_2 = nn.LayerNorm()
        self.dropout = nn.Dropout(self.dropout_prob)

    def __call__(self, x, train=True):
        inp_x = self.layer_norm_1(x)
        attn_out = self.attn(inputs_q=inp_x, inputs_kv=inp_x)
        x = x + self.dropout(attn_out, deterministic=not train)

        linear_out = self.layer_norm_2(x)
        for l in self.linear:
            linear_out = l(linear_out) if not isinstance(l, nn.Dropout) else l(linear_out, deterministic=not train)
```

(continues on next page)

(continued from previous page)

```
x = x + self.dropout(linear_out, deterministic=not train)
return x
```

```
[8]: ## Test AttentionBlock implementation
# Example features as input
main_rng, x_rng = random.split(main_rng)
x = random.normal(x_rng, (3, 16, 128))
# Create attention block
attnblock = AttentionBlock(embed_dim=128, hidden_dim=512, num_heads=4, dropout_prob=0.1)
# Initialize parameters of attention block with random key and inputs
main_rng, init_rng, dropout_init_rng = random.split(main_rng, 3)
params = attnblock.init({'params': init_rng, 'dropout': dropout_init_rng}, x, True)[
    'params']
# Apply encoder block with parameters on the inputs
# Since dropout is stochastic, we need to pass a rng to the forward
main_rng, dropout_apply_rng = random.split(main_rng)
out = attnblock.apply({'params': params}, x, train=True, rands={'dropout': dropout_apply_
    _rng})
print('Out', out.shape)

del attnblock, params
Out (3, 16, 128)
```

Now we have all modules ready to build our own Vision Transformer. Besides the Transformer encoder, we need the following modules:

- A **linear projection** layer that maps the input patches to a feature vector of larger size. It is implemented by a simple linear layer that takes each $M \times M$ patch independently as input.
- A **classification token** that is added to the input sequence. We will use the output feature vector of the classification token (CLS token in short) for determining the classification prediction.
- Learnable **positional encodings** that are added to the tokens before being processed by the Transformer. Those are needed to learn position-dependent information, and convert the set to a sequence. Since we usually work with a fixed resolution, we can learn the positional encodings instead of having the pattern of sine and cosine functions.
- An **MLP head** that takes the output feature vector of the CLS token, and maps it to a classification prediction. This is usually implemented by a small feed-forward network or even a single linear layer.

With those components in mind, let's implement the full Vision Transformer below:

```
[9]: class VisionTransformer(nn.Module):
    embed_dim : int      # Dimensionality of input and attention feature vectors
    hidden_dim : int     # Dimensionality of hidden layer in feed-forward network
    num_heads : int      # Number of heads to use in the Multi-Head Attention block
    num_channels : int   # Number of channels of the input (3 for RGB)
    num_layers : int     # Number of layers to use in the Transformer
    num_classes : int    # Number of classes to predict
    patch_size : int     # Number of pixels that the patches have per dimension
    num_patches : int    # Maximum number of patches an image can have
    dropout_prob : float = 0.0 # Amount of dropout to apply in the feed-forward network

    def setup(self):
```

(continues on next page)

(continued from previous page)

```

# Layers/Networks
self.input_layer = nn.Dense(self.embed_dim)
self.transformer = [AttentionBlock(self.embed_dim,
                                  self.hidden_dim,
                                  self.num_heads,
                                  self.dropout_prob) for _ in range(self.num_
→layers)]
self.mlp_head = nn.Sequential([
    nn.LayerNorm(),
    nn.Dense(self.num_classes)
])
self.dropout = nn.Dropout(self.dropout_prob)

# Parameters/Embeddings
self.cls_token = self.param('cls_token',
                            nn.initializers.normal(stddev=1.0),
                            (1, 1, self.embed_dim))
self.pos_embedding = self.param('pos_embedding',
                                nn.initializers.normal(stddev=1.0),
                                (1, 1+self.num_patches, self.embed_dim))

def __call__(self, x, train=True):
    # Preprocess input
    x = img_to_patch(x, self.patch_size)
    B, T, _ = x.shape
    x = self.input_layer(x)

    # Add CLS token and positional encoding
    cls_token = self.cls_token.repeat(B, axis=0)
    x = jnp.concatenate([cls_token, x], axis=1)
    x = x + self.pos_embedding[:, :T+1]

    # Apply Transformer
    x = self.dropout(x, deterministic=not train)
    for attn_block in self.transformer:
        x = attn_block(x, train=train)

    # Perform classification prediction
    cls = x[:, 0]
    out = self.mlp_head(cls)
    return out

```

[10]: ## Test VisionTransformer implementation

```

# Example features as input
main_rng, x_rng = random.split(main_rng)
x = random.normal(x_rng, (5, 32, 32, 3))
# Create vision transformer
visntrans = VisionTransformer(embed_dim=128,
                               hidden_dim=512,
                               num_heads=4,
                               num_channels=3,

```

(continues on next page)

(continued from previous page)

```

    num_layers=6,
    num_classes=10,
    patch_size=4,
    num_patches=64,
    dropout_prob=0.1)
# Initialize parameters of the Vision Transformer with random key and inputs
main_rng, init_rng, dropout_init_rng = random.split(main_rng, 3)
params = visntrans.init({'params': init_rng, 'dropout': dropout_init_rng}, x, True)[
    ↪'params']
# Apply encoder block with parameters on the inputs
# Since dropout is stochastic, we need to pass a rng to the forward
main_rng, dropout_apply_rng = random.split(main_rng)
out = visntrans.apply({'params': params}, x, train=True, rands={'dropout': dropout_apply_-
    ↪rng})
print('Out', out.shape)

def visntrans, params
Out (5, 10)

```

Finally, we can put everything into a trainer module as usual. We use `optax.adamw` as the optimizer, which is Adam with a corrected weight decay implementation. Since we use the Pre-LN Transformer version, we do not need to use a learning rate warmup stage anymore. Instead, we use the same learning rate scheduler as the CNNs in our previous tutorial on image classification.

```

[11]: class TrainerModule:

    def __init__(self, exmp_imgs, lr=1e-3, weight_decay=0.01, seed=42, **model_hparams):
        """
        Module for summarizing all training functionalities for classification on
        ↪CIFAR10.

        Inputs:
            exmp_imgs - Example imgs, used as input to initialize the model
            lr - Learning rate of the optimizer to use
            weight_decay - Weight decay to use in the optimizer
            seed - Seed to use in the model initialization
        """
        super().__init__()
        self.lr = lr
        self.weight_decay = weight_decay
        self.seed = seed
        self.rng = jax.random.PRNGKey(self.seed)
        # Create empty model. Note: no parameters yet
        self.model = VisionTransformer(**model_hparams)
        # Prepare logging
        self.log_dir = os.path.join(CHECKPOINT_PATH, 'ViT/')
        self.logger = SummaryWriter(log_dir=self.log_dir)
        # Create jitted training and eval functions
        self.create_functions()
        # Initialize model
        self.init_model(exmp_imgs)

```

(continues on next page)

(continued from previous page)

```

def create_functions(self):
    # Function to calculate the classification loss and accuracy for a model
    def calculate_loss(params, rng, batch, train):
        imgs, labels = batch
        rng, dropout_apply_rng = random.split(rng)
        logits = self.model.apply({'params': params},
                                  imgs,
                                  train=train,
                                  rngs={'dropout': dropout_apply_rng})
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, labels).mean()
        acc = (logits.argmax(axis=-1) == labels).mean()
        return loss, (acc, rng)

    # Training function
    def train_step(state, rng, batch):
        loss_fn = lambda params: calculate_loss(params, rng, batch, train=True)
        # Get loss, gradients for loss, and other outputs of loss function
        (loss, (acc, rng)), grads = jax.value_and_grad(loss_fn, has_aux=True)(state.
        ↪params)
        # Update parameters and batch statistics
        state = state.apply_gradients(grads=grads)
        return state, rng, loss, acc

    # Eval function
    def eval_step(state, rng, batch):
        # Return the accuracy for a single batch
        _, (acc, rng) = calculate_loss(state.params, rng, batch, train=False)
        return rng, acc

    # jit for efficiency
    self.train_step = jax.jit(train_step)
    self.eval_step = jax.jit(eval_step)

    def init_model(self, exmp_imgs):
        # Initialize model
        self.rng, init_rng, dropout_init_rng = random.split(self.rng, 3)
        self.init_params = self.model.init({'params': init_rng, 'dropout': dropout_init_
        ↪rng}),
                                         exmp_imgs,
                                         train=True)['params']
        self.state = None

    def init_optimizer(self, num_epochs, num_steps_per_epoch):
        # We decrease the learning rate by a factor of 0.1 after 60% and 85% of the_
        ↪training
        lr_schedule = optax.piecewise_constant_schedule(
            init_value=self.lr,
            boundaries_and_scales=
                {int(num_steps_per_epoch * num_epochs * 0.6): 0.1,
                 int(num_steps_per_epoch * num_epochs * 0.85): 0.1}
        )
        optimizer = optax.chain(
            optax.clip_by_global_norm(1.0), # Clip gradients at norm 1
            optax.adamw(lr_schedule, weight_decay=self.weight_decay)
        )

```

(continues on next page)

(continued from previous page)

```

# Initialize training state
self.state = train_state.TrainState.create(
    apply_fn=self.model.apply,
    params=self.init_params if self.state is None
else self.state.params,
    tx=optimizer)

def train_model(self, train_loader, val_loader, num_epochs=200):
    # Train model for defined number of epochs
    # We first need to create optimizer and the scheduler for the given number of
epochs
    self.init_optimizer(num_epochs, len(train_loader))
    # Track best eval accuracy
    best_eval = 0.0
    for epoch_idx in tqdm(range(1, num_epochs+1)):
        self.train_epoch(epoch=epoch_idx)
        if epoch_idx % 2 == 0:
            eval_acc = self.eval_model(val_loader)
            self.logger.add_scalar('val/acc', eval_acc, global_step=epoch_idx)
            if eval_acc >= best_eval:
                best_eval = eval_acc
                self.save_model(step=epoch_idx)
            self.logger.flush()

def train_epoch(self, epoch):
    # Train model for one epoch, and log avg loss and accuracy
    metrics = defaultdict(list)
    for batch in tqdm(train_loader, desc='Training', leave=False):
        self.state, self.rng, loss, acc = self.train_step(self.state, self.rng,
batch)
        metrics['loss'].append(loss)
        metrics['acc'].append(acc)
    for key in metrics:
        avg_val = np.stack(jax.device_get(metrics[key])).mean()
        self.logger.add_scalar('train/' + key, avg_val, global_step=epoch)

def eval_model(self, data_loader):
    # Test model on all images of a data loader and return avg loss
    correct_class, count = 0, 0
    for batch in data_loader:
        self.rng, acc = self.eval_step(self.state, self.rng, batch)
        correct_class += acc * batch[0].shape[0]
        count += batch[0].shape[0]
    eval_acc = (correct_class / count).item()
    return eval_acc

def save_model(self, step=0):
    # Save current model at certain training iteration
    checkpoints.save_checkpoint(ckpt_dir=self.log_dir,
                                target=self.state.params,
                                step=step,
                                overwrite=True)

```

(continues on next page)

(continued from previous page)

```

def load_model(self, pretrained=False):
    # Load model. We use different checkpoint for pretrained models
    if not pretrained:
        params = checkpoints.restore_checkpoint(ckpt_dir=self.log_dir, target=None)
    else:
        params = checkpoints.restore_checkpoint(ckpt_dir=os.path.join(CHECKPOINT_PATH,
            f'ViT.ckpt'), target=None)
    self.state = train_state.TrainState.create(
        apply_fn=self.model.apply,
        params=params,
        tx=self.state.tx if self.state else optax.
        adamw(self.lr) # Default optimizer
    )

def checkpoint_exists(self):
    # Check whether a pretrained model exist for this autoencoder
    return os.path.isfile(os.path.join(CHECKPOINT_PATH, f'ViT.ckpt'))

```

4.28.2 Experiments

Commonly, Vision Transformers are applied to large-scale image classification benchmarks such as ImageNet to leverage their full potential. However, here we take a step back and ask: can Vision Transformer also succeed on classical, small benchmarks such as CIFAR10? To find this out, we train a Vision Transformer from scratch on the CIFAR10 dataset. Let's first create a training function for our PyTorch Lightning module which also loads the pre-trained model if you have downloaded it above.

```
[12]: def train_model(*args, num_epochs=200, **kwargs):
    # Create a trainer module with specified hyperparameters
    trainer = TrainerModule(*args, **kwargs)
    if not trainer.checkpoint_exists(): # Skip training if pretrained model exists
        trainer.train_model(train_loader, val_loader, num_epochs=num_epochs)
        trainer.load_model()
    else:
        trainer.load_model(pretrained=True)
    # Test trained model
    val_acc = trainer.eval_model(val_loader)
    test_acc = trainer.eval_model(test_loader)
    return trainer, {'val': val_acc, 'test': test_acc}
```

Now, we can already start training our model. As seen in our implementation, we have a couple of hyperparameters that we have to set. When creating this notebook, we have performed a small grid search over hyperparameters and listed the best hyperparameters in the cell below. Nevertheless, it is worth discussing the influence that each hyperparameter has, and what intuition we have for choosing its value.

First, let's consider the patch size. The smaller we make the patches, the longer the input sequences to the Transformer become. While in general, this allows the Transformer to model more complex functions, it requires a longer computation time due to its quadratic memory usage in the attention layer. Furthermore, small patches can make the task more difficult since the Transformer has to learn which patches are close-by, and which are far away. We experimented with patch sizes of 2, 4, and 8 which gives us the input sequence lengths of 256, 64, and 16 respectively. We found 4 to result in the best performance and hence pick it below.

Next, the embedding and hidden dimensionality have a similar impact on a Transformer as to an MLP. The larger the

sizes, the more complex the model becomes, and the longer it takes to train. In Transformers, however, we have one more aspect to consider: the query-key sizes in the Multi-Head Attention layers. Each key has the feature dimensionality of `embed_dim/num_heads`. Considering that we have an input sequence length of 64, a minimum reasonable size for the key vectors is 16 or 32. Lower dimensionalities can restrain the possible attention maps too much. We observed that more than 8 heads are not necessary for the Transformer, and therefore pick an embedding dimensionality of 256. The hidden dimensionality in the feed-forward networks is usually 2-4x larger than the embedding dimensionality, and thus we pick 512.

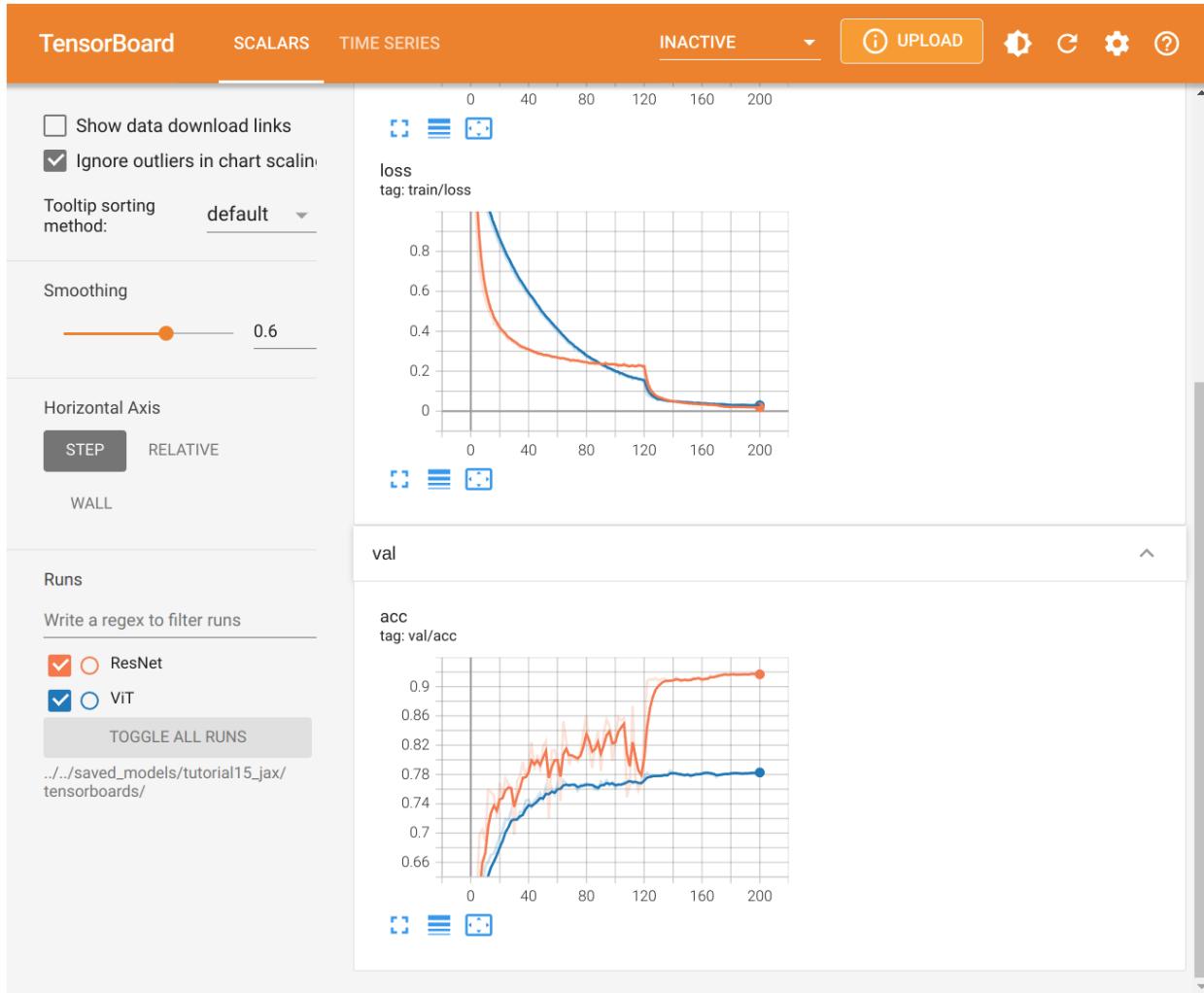
Finally, the learning rate for Transformers is usually relatively small, and in papers, a common value to use is `3e-5`. However, since we work with a smaller dataset and have a potentially easier task, we found that we are able to increase the learning rate to `3e-4` without any problems. To reduce overfitting, we use a dropout value of 0.2. Remember that we also use small image augmentations as regularization during training.

Feel free to explore the hyperparameters yourself by changing the values below. In general, the Vision Transformer did not show to be too sensitive to the hyperparameter choices on the CIFAR10 dataset.

```
[13]: model, results = train_model(exmp_imgs=next(iter(train_loader))[0],
                                 embed_dim=256,
                                 hidden_dim=512,
                                 num_heads=8,
                                 num_layers=6,
                                 patch_size=4,
                                 num_channels=3,
                                 num_patches=64,
                                 num_classes=10,
                                 dropout_prob=0.2,
                                 lr=3e-4)
print("ViT results", results)
ViT results {'val': 0.7856000661849976, 'test': 0.784000039100647}
```

The Vision Transformer achieves a validation and test performance of about 78%. In comparison, almost all CNN architectures that we have tested in [Tutorial 5](#) obtained a classification performance of around 90%. This is a considerable gap and shows that although Vision Transformers perform strongly on ImageNet with potential pretraining, they cannot come close to simple CNNs on CIFAR10 when being trained from scratch. The differences between a CNN and Transformer can be well observed in the training curves. Let's look at them in a tensorboard below:

```
[14]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH!
%tensorboard --logdir ../../saved_models/tutorial15_jax/tensorboards/
```



The tensorboard compares the Vision Transformer to a ResNet trained on CIFAR10. When looking at the training losses, we see that the ResNet learns much more quickly in the first iterations. While the learning rate might have an influence on the initial learning speed, we see the same trend in the validation accuracy. The ResNet achieves the best performance of the Vision Transformer after just 5 epochs (2000 iterations). Further, while the ResNet training loss and validation accuracy have a similar trend, the validation performance of the Vision Transformers only marginally changes after 10k iterations while the training loss has almost just started going down. Yet, the Vision Transformer is also able to achieve close to 100% accuracy on the training set.

All those observed phenomena can be explained with a concept that we have visited before: inductive biases. Convolutional Neural Networks have been designed with the assumption that images are translation invariant. Hence, we apply convolutions with shared filters across the image. Furthermore, a CNN architecture integrates the concept of distance in an image: two pixels that are close to each other are more related than two distant pixels. Local patterns are combined into larger patterns until we perform our classification prediction. All those aspects are inductive biases of a CNN. In contrast, a Vision Transformer does not know which two pixels are close to each other, and which are far apart. It has to learn this information solely from the sparse learning signal of the classification task. This is a huge disadvantage when we have a small dataset since such information is crucial for generalizing to an unseen test dataset. With large enough datasets and/or good pre-training, a Transformer can learn this information without the need for inductive biases, and instead is more flexible than a CNN. Especially long-distance relations between local patterns can be difficult to process in CNNs, while in Transformers, all patches have the distance of one. This is why Vision Transformers are so strong on large-scale datasets such as ImageNet but underperform a lot when being applied to a small dataset such as CIFAR10.

4.28.3 Conclusion

In this tutorial, we have implemented our own Vision Transformer from scratch and applied it to the task of image classification. Vision Transformers work by splitting an image into a sequence of smaller patches, use those as input to a standard Transformer encoder. While Vision Transformers achieved outstanding results on large-scale image recognition benchmarks such as ImageNet, they considerably underperform when being trained from scratch on small-scale datasets like CIFAR10. The reason is that in contrast to CNNs, Transformers do not have the inductive biases of translation invariance and the feature hierarchy (i.e. larger patterns consist of many smaller patterns). However, these aspects can be learned when enough data is provided, or the model has been pre-trained on other large-scale tasks. Considering that Vision Transformers have just been proposed end of 2020, there is likely a lot more to come on Transformers for Computer Vision.

References

- Dosovitskiy, Alexey, et al. “An image is worth 16x16 words: Transformers for image recognition at scale.” International Conference on Learning Representations (2021). [link](#)
- Chen, Xiangning, et al. “When Vision Transformers Outperform ResNets without Pretraining or Strong Data Augmentations.” arXiv preprint arXiv:2106.01548 (2021). [link](#)
- Tolstikhin, Ilya, et al. “MLP-mixer: An all-MLP Architecture for Vision.” arXiv preprint arXiv:2105.01601 (2021). [link](#)
- Xiong, Ruibin, et al. “On layer normalization in the transformer architecture.” International Conference on Machine Learning. PMLR, 2020. [link](#)

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.29 Tutorial 17 (JAX): Self-Supervised Contrastive Learning with SimCLR

Filled notebook:

Pre-trained models:

PyTorch version:

Author: Phillip Lippe

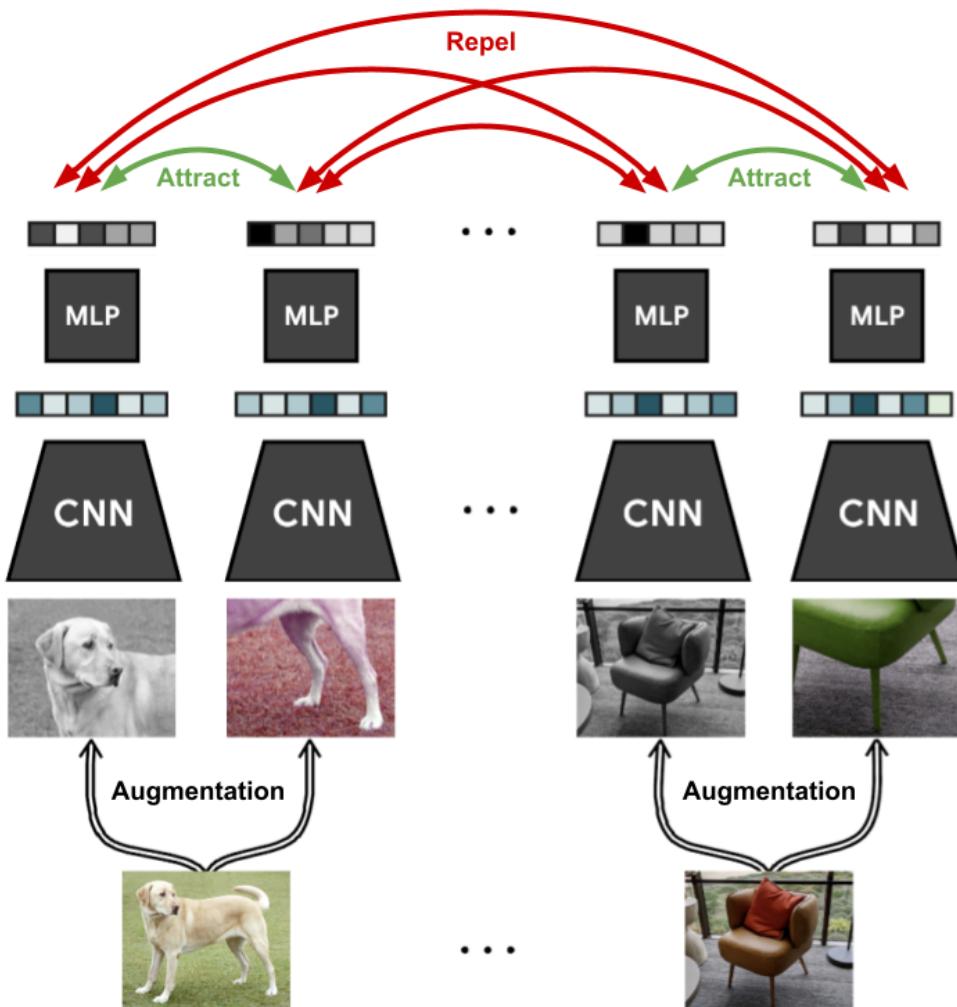
Note: This notebook is written in JAX+Flax. It is a 1-to-1 translation of the original notebook written in PyTorch+PyTorch Lightning with almost identical results. For an introduction to JAX, check out our [Tutorial 2 \(JAX\): Introduction to JAX+Flax](#). Further, throughout the notebook, we comment on major differences to the PyTorch version and provide explanations for the major parts of the JAX code.

Speed comparison: We will report the speed comparisons of the PyTorch and JAX implementation soon.

Models	PyTorch	JAX
SimCLR	-min -sec	-min -sec
Logistic Regression	-min -sec	-min -sec
Supervised ResNet-18	-min -sec	-min -sec

In this tutorial, we will take a closer look at self-supervised contrastive learning. Self-supervised learning, or also sometimes called unsupervised learning, describes the scenario where we have given input data, but no accompanying labels to train in a classical supervised way. However, this data still contains a lot of information from which we can learn: how are the images different from each other? What patterns are descriptive for certain images? Can we cluster the images? And so on. Methods for self-supervised learning try to learn as much as possible from the data alone, so it can quickly be finetuned for a specific classification task. The benefit of self-supervised learning is that a large dataset can often easily be obtained. For instance, if we want to train a vision model on semantic segmentation for autonomous driving, we can collect large amounts of data by simply installing a camera in a car, and driving through a city for an hour. In contrast, if we would want to do supervised learning, we would have to manually label all those images before training a model. This is extremely expensive, and would likely take a couple of months to manually label the same amount of data. Further, self-supervised learning can provide an alternative to transfer learning from models pretrained on ImageNet since we could pretrain a model on a specific dataset/situation, e.g. traffic scenarios for autonomous driving.

Within the last two years, a lot of new approaches have been proposed for self-supervised learning, in particular for images, that have resulted in great improvements over supervised models when few labels are available. The subfield that we will focus on in this tutorial is contrastive learning. Contrastive learning is motivated by the question mentioned above: how are images different from each other? Specifically, contrastive learning methods train a model to cluster an image and its slightly augmented version in latent space, while the distance to other images should be maximized. A very recent and simple method for this is [SimCLR](#), which is visualized below (figure credit - [Ting Chen et al.](#)).



The general setup is that we are given a dataset of images without any labels, and want to train a model on this data such that it can quickly adapt to any image recognition task afterward. During each training iteration, we sample a batch of images as usual. For each image, we create two versions by applying data augmentation techniques like cropping, Gaussian noise, blurring, etc. An example of such is shown on the left with the image of the dog. We will go into the details and effects of the chosen augmentation techniques later. On those images, we apply a CNN like ResNet and obtain as output a 1D feature vector on which we apply a small MLP. The output features of the two augmented images are then trained to be close to each other, while all other images in that batch should be as different as possible. This way, the model has to learn to recognize the content of the image that remains unchanged under the data augmentations, such as objects which we usually care about in supervised tasks.

We will now implement this framework ourselves and discuss further details along the way. Let's first start with importing our standard libraries below:

```
[1]: ## Standard libraries
import os
import numpy as np
from typing import Sequence, Any
from collections import defaultdict

## Imports for plotting
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.set()

## tqdm for loading bars
from tqdm.auto import tqdm

## To run JAX on TPU in Google Colab, uncomment the two lines below
# import jax.tools.colab_tpu
# jax.tools.colab_tpu.setup_tpu()

## JAX
import jax
import jax.numpy as jnp
from jax import random

## Flax (NN in JAX)
try:
    import flax
except ModuleNotFoundError: # Install flax if missing
    !pip install --quiet flax
    import flax
from flax import linen as nn
from flax.training import train_state, checkpoints

## Optax (Optimizers in JAX)
try:
    import optax
except ModuleNotFoundError: # Install optax if missing
    !pip install --quiet optax
    import optax

## PyTorch
import torch
import torch.utils.data as data
from torch.utils.tensorboard import SummaryWriter
import torchvision
from torchvision import transforms
from torchvision.datasets import STL10

# Import tensorboard
%load_ext tensorboard

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../../saved_models/tutorial17_jax"
```

(continues on next page)

(continued from previous page)

```
print('Device:', jax.devices()[0])
Device: gpu:0
```

As in many tutorials before, we provide pre-trained models. Note that those models are slightly larger as normal (~100MB overall) since we use the default ResNet-18 architecture. If you are running this notebook locally, make sure to have sufficient disk space available.

```
[2]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/JAX/tutorial17/"
# Files to download
pretrained_files = ["SimCLR.ckpt", "ResNet.ckpt",
                    "tensorboards/SimCLR/events.out.tfevents.SimCLR",
                    "tensorboards/classification/ResNet/events.out.tfevents.ResNet"]
pretrained_files += [f"LogisticRegression_{size}.ckpt" for size in [10, 20, 50, 100, 200,
                                                               500]]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/", 1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive_"
                  "folder, or contact the author with the full output including the following error:\n", e)
```

4.29.1 SimCLR

We will start our exploration of contrastive learning by discussing the effect of different data augmentation techniques, and how we can implement an efficient data loader for such, specialized for JAX. Next, we implement SimCLR with Flax, and finally train it on a large, unlabeled dataset.

Data Augmentation for Contrastive Learning

To allow efficient training, we need to prepare the data loading such that we sample two different, random augmentations for each image in the batch. The easiest way to do this is by creating a transformation that, when being called, applies a set of data augmentations to an image twice. This is implemented in the class `ContrastiveTransformations` below:

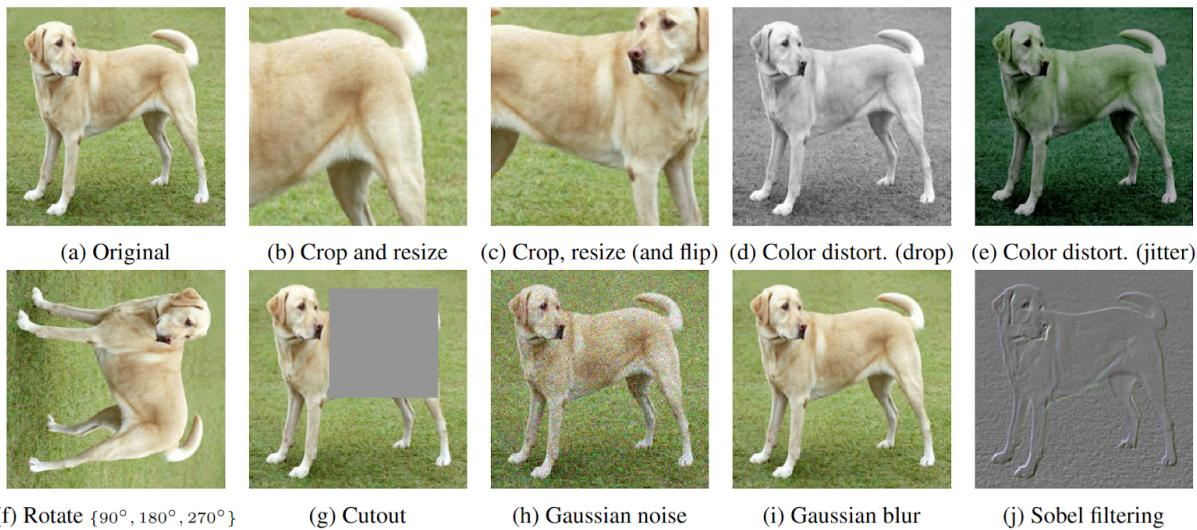
```
[3]: class ContrastiveTransformations(object):

    def __init__(self, base_transforms, n_views=2):
        self.base_transforms = base_transforms
        self.n_views = n_views

    def __call__(self, x):
        return [self.base_transforms(x) for i in range(self.n_views)]
```

The contrastive learning framework can easily be extended to have more *positive* examples by sampling more than two augmentations of the same image. However, the most efficient training is usually obtained by using only two.

Next, we can look at the specific augmentations we want to apply. The choice of the data augmentation to use is the most crucial hyperparameter in SimCLR since it directly affects how the latent space is structured, and what patterns might be learned from the data. Let's first take a look at some of the most popular data augmentations (figure credit - Ting Chen and Geoffrey Hinton):



All of them can be used, but it turns out that two augmentations stand out in their importance: crop-and-resize, and color distortion. Interestingly, however, they only lead to strong performance if they have been used together as discussed by Ting Chen et al. in their SimCLR paper. When performing randomly cropping and resizing, we can distinguish between two situations: (a) cropped image A provides a local view of cropped image B, or (b) cropped images C and D show neighboring views of the same image (figure credit - Ting Chen and Geoffrey Hinton).

While situation (a) requires the model to learn some sort of scale invariance to make crops A and B similar in latent space, situation (b) is more challenging since the model needs to recognize an object beyond its limited view. However, without color distortion, there is a loophole that the model can exploit, namely that different crops of the same image usually look very similar in color space. Consider the picture of the dog above. Simply from the color of the fur and the green color tone of the background, you can reason that two patches belong to the same image without actually recognizing the dog in the picture. In this case, the model might end up focusing only on the color histograms of the images, and ignore other more generalizable features. If, however, we distort the colors in the two patches randomly

and independently of each other, the model cannot rely on this simple feature anymore. Hence, by combining random cropping and color distortions, the model can only match two patches by learning generalizable representations.

Overall, for our experiments, we apply a set of 5 transformations following the original SimCLR setup: random horizontal flip, crop-and-resize, color distortion, random grayscale, and gaussian blur. In comparison to the [original implementation](#), we reduce the effect of the color jitter slightly (0.5 instead of 0.8 for brightness, contrast, and saturation, and 0.1 instead of 0.2 for hue). In our experiments, this setting obtained better performance and was faster and more stable to train. If, for instance, the brightness scale highly varies in a dataset, the original settings can be more beneficial since the model can't rely on this information anymore to distinguish between images.

Image Augmentations with JAX

In the [PyTorch version](#) of this tutorial, we implement all these augmentations with transformation from the `torchvision` library. This requires a considerable amount of compute from the CPU to prepare the data in time for using it on the GPU. As alternative, we show here an implementation with JAX-native transformations in `Pix` that are executed on GPU on batch-level, using `jax.vmap` and `jax.jit`. Still, one transformation that can be difficult in JAX is resize-and-crop, since the resize creates arrays of dynamic shapes. Hence, we use the `RandomResizedCrop` transformation from `torchvision`:

```
[4]: def image_to_numpy(img):
    img = np.array(img, dtype=np.float32)
    img = img / 255.
    return img

contrast_transforms = transforms.Compose([transforms.RandomResizedCrop(size=96),
                                         image_to_numpy])
```

`Pix` is an image-processing library in JAX, developed by Deep mind, that implements most common data augmentations. Let's first import it below:

```
[5]: try:
    import dm_pix
except ModuleNotFoundError: # Install pix if missing
    !pip install --quiet dm_pix
    import dm_pix
```

As a first step, we implement a function that augments a single image and applies a random horizontal flip, color jitter (brightness, contrast, saturation, and hue), random grayscaling, and gaussian blur. Since these are operations with stochasticity/randomness, we also pass a random seed with it that we can split for each operation.

```
[6]: def augment_image(rng, img):
    rngs = random.split(rng, 8)
    # Random left-right flip
    img = dm_pix.random_flip_left_right(rngs[0], img)
    # Color jitter
    img_jt = img
    img_jt = img_jt * random.uniform(rngs[1], shape=(1,), minval=0.5, maxval=1.5) # Brightness
    img_jt = jax.lax.clamp(0.0, img_jt, 1.0)
    img_jt = dm_pix.random_contrast(rngs[2], img_jt, lower=0.5, upper=1.5)
    img_jt = jax.lax.clamp(0.0, img_jt, 1.0)
    img_jt = dm_pix.random_saturation(rngs[3], img_jt, lower=0.5, upper=1.5)
    img_jt = jax.lax.clamp(0.0, img_jt, 1.0)
```

(continues on next page)

(continued from previous page)

```



```

You might have noticed that we do not use the function `dm_pix.random_brightness` for the brightness augmentation. The reason is that the implementation follows TensorFlow's brightness augmentation by adding a constant value to each pixel, while the implementation in torchvision and PIL scales the pixel values by the factor. To stay closer to the PyTorch implementation, we implement it ourselves by scaling the pixel values.

Now that we have a function to augment a single image, let's transform this function to allow for a batch of images. This can be done using `jax.vmap`, with which we can vectorize the function both over images and seeds, such that each image is differently augmented. To not have to pass N seeds for a batch of N images, we can simply split one seed N times before calling the augmentation function. Finally, let's optimize the whole augmentation pipeline by just-in-time compilation:

```
[7]: parallel_augment = jax.jit(lambda rng, imgs: jax.vmap(augment_image)(random.split(rng, imgs.shape[0])), static_argnames=['rng'])
```

In contrast to the PyTorch implementation, this augmentation will be performed on GPU, integrated in the training step for contrastive learning. The benefit of this is that we require much less CPU compute power and don't have to worry much about having enough workers etc. in the data loader. However, the apparent disadvantage is that we strictly block the GPU for the augmentations. But will this even be noticeable in practice? We can easily check that by timing the execution for our augmentations on an arbitrary batch of images:

```
[8]: img_rng, augm_rng = random.split(random.PRNGKey(42))
rand_imgs = random.uniform(img_rng, (512, 96, 96, 3))
_ = parallel_augment(augm_rng, rand_imgs)
```

```
[9]: %timeit
_ = parallel_augment(augm_rng, rand_imgs)
8.3 ms ± 4.55 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Even for this very large batch of images, the augmentation takes only a few milliseconds on a common GPU, while a training step of a ResNet-18, as we will see later, takes several magnitudes more (e.g. about 200ms on an NVIDIA GTX1080Ti). Hence, we will not really notice this extra computation during training, while being much more efficient in CPU usage.

STL10 dataset

After discussing the data augmentation techniques, we can now focus on the dataset. In this tutorial, we will use the [STL10 dataset](#), which, similarly to CIFAR10, contains images of 10 classes: airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck. However, the images have a higher resolution, namely 96×96 pixels, and we are only provided with 500 labeled images per class. Additionally, we have a much larger set of 100,000 unlabeled images which are similar to the training images but are sampled from a wider range of animals and vehicles. This makes the dataset ideal to showcase the benefits that self-supervised learning offers.

Luckily, the STL10 dataset is provided through torchvision. Keep in mind, however, that since this dataset is relatively large and has a considerably higher resolution than CIFAR10, it requires more disk space (~3GB) and takes a bit of time to download. For our initial discussion of self-supervised learning and SimCLR, we will create two data loaders with our contrastive transformations above: the `unlabeled_data` will be used to train our model via contrastive learning, and `train_data_contrast` will be used as a validation set in contrastive learning.

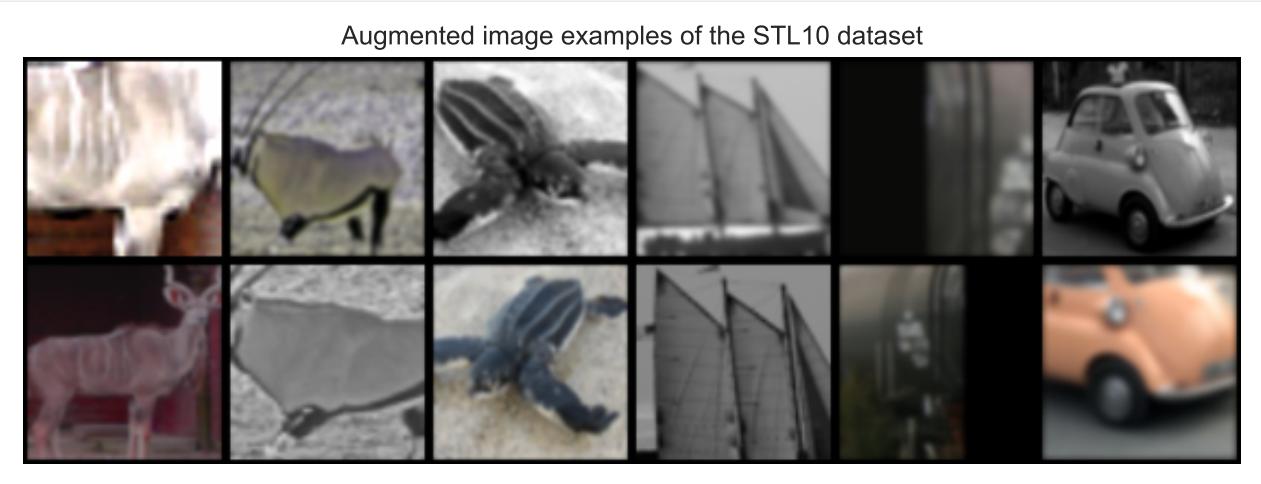
```
[10]: unlabeled_data = STL10(root=DATASET_PATH, split='unlabeled', download=True,
                           transform=ContrastiveTransformations(contrast_transforms, n_
                           ↵views=2))
train_data_contrast = STL10(root=DATASET_PATH, split='train', download=True,
                           transform=ContrastiveTransformations(contrast_transforms, n_
                           ↵views=2))

Files already downloaded and verified
Files already downloaded and verified
```

Finally, before starting with our implementation of SimCLR, let's look at some example image pairs sampled with our augmentations:

```
[11]: # Visualize some examples
NUM_IMAGES = 6
imgs = np.stack([parallel_augment(random.PRNGKey(idx), np.stack(unlabeled_data[idx][0], axis=0)) for idx in range(NUM_IMAGES)], axis=2)
imgs = (imgs + 1.0) / 2.0
img_grid = np.pad(imgs, ((0,0), (2,2), (0,0), (2,2), (0,0)))
img_grid = img_grid.reshape(img_grid.shape[0]*img_grid.shape[1], -1, img_grid.shape[-1])

plt.figure(figsize=(10,5))
plt.title('Augmented image examples of the STL10 dataset')
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```



We see the wide variety of our data augmentation, including randomly cropping, grayscaling, gaussian blur, and color distortion. Thus, it remains a challenging task for the model to match two, independently augmented patches of the same image.

SimCLR implementation

Using the data loader pipeline above, we can now implement SimCLR. At each iteration, we get for every image x two differently augmented versions, which we refer to as \tilde{x}_i and \tilde{x}_j . Both of these images are encoded into a one-dimensional feature vector, between which we want to maximize similarity which minimizes it to all other images in the batch. The encoder network is split into two parts: a base encoder network $f(\cdot)$, and a projection head $g(\cdot)$. The base network is usually a deep CNN as we have seen in e.g. [Tutorial 5](#) before, and is responsible for extracting a representation vector from the augmented data examples. In our experiments, we will use the common ResNet-18 architecture as $f(\cdot)$, and refer to the output as $f(\tilde{x}_i) = h_i$. The projection head $g(\cdot)$ maps the representation h into a space where we apply the contrastive loss, i.e., compare similarities between vectors. It is often chosen to be a small MLP with non-linearities, and for simplicity, we follow the original SimCLR paper setup by defining it as a two-layer MLP with ReLU activation in the hidden layer. Note that in the follow-up paper, [SimCLRV2](#), the authors mention that larger/wider MLPs can boost the performance considerably. This is why we apply an MLP with four times larger hidden dimensions, but deeper MLPs showed to overfit on the given dataset. The general setup is visualized below (figure credit - [Ting Chen et al.](#)):

After finishing the training with contrastive learning, we will remove the projection head $g(\cdot)$, and use $f(\cdot)$ as a pre-trained feature extractor. The representations z that come out of the projection head $g(\cdot)$ have been shown to perform worse than those of the base network $f(\cdot)$ when finetuning the network for a new task. This is likely because the representations z are trained to become invariant to many features like the color that can be important for downstream tasks. Thus, $g(\cdot)$ is only needed for the contrastive learning stage.

Now that the architecture is described, let's take a closer look at how we train the model. As mentioned before, we want to maximize the similarity between the representations of the two augmented versions of the same image, i.e., z_i and z_j in the figure above, while minimizing it to all other examples in the batch. SimCLR thereby applies the InfoNCE loss, originally proposed by [Aaron van den Oord et al.](#) for contrastive learning. In short, the InfoNCE loss compares the similarity of z_i and z_j to the similarity of z_i to any other representation in the batch by performing a softmax over the similarity values. The loss can be formally written as:

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(z_i, z_k)/\tau)} = -\text{sim}(z_i, z_j)/\tau + \log \left[\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(z_i, z_k)/\tau) \right]$$

The function `sim` is a similarity metric, and the hyperparameter τ is called temperature determining how peaked the distribution is. Since many similarity metrics are bounded, the temperature parameter allows us to balance the influence of many dissimilar image patches versus one similar patch. The similarity metric that is used in SimCLR is cosine similarity, as defined below:

$$\text{sim}(z_i, z_j) = \frac{z_i^\top \cdot z_j}{\|z_i\| \cdot \|z_j\|}$$

The maximum cosine similarity possible is 1, while the minimum is -1 . In general, we will see that the features of two different images will converge to a cosine similarity around zero since the minimum, -1 , would require z_i and z_j to be in the exact opposite direction in all feature dimensions, which does not allow for great flexibility.

Finally, now that we have discussed all details, let's implement SimCLR below as a Flax module. To not implement ResNet-18 ourselves, we use `flaxmodels` as we have already done in Tutorial 6.

```
[12]: # Import and install flaxmodels if needed
try:
    import flaxmodels
except ModuleNotFoundError:
    !pip install --upgrade git+https://github.com/matthias-wright/flaxmodels.git
    import flaxmodels
```

```
[13]: class SimCLR(nn.Module):
    hidden_dim : int
    temperature : float

    def setup(self):
        # Base model f(.) - ResNet 18 with last layer being size of 4*hidden_dim
        self.convnet = flaxmodels.ResNet18(output='activations',
                                           pretrained=False,
                                           normalize=False,
                                           num_classes=4*self.hidden_dim)
        # Network g(.) as MLP with last fc layer of convnet
        self.head = nn.Sequential([
            nn.relu,
            nn.Dense(self.hidden_dim)
        ])

    def __call__(self, imgs, train=True):
        # Encode all images
        model_feats = self.convnet(imgs, train=train)
        feats = self.head(model_feats['fc'])

        # Calculate cosine similarity between all images
        cos_sim = optax.cosine_similarity(feats[:,None,:,:], feats[None,:,:,:])
        cos_sim /= self.temperature
        # Masking cosine similarities to itself
        diag_range = jnp.arange(feats.shape[0], dtype=jnp.int32)
        cos_sim = cos_sim.at[diag_range, diag_range].set(-9e15)
        # Find positive example -> batch_size//2 away from the original example
        shifted_diag = jnp.roll(diag_range, imgs.shape[0]//2)
        pos_logits = cos_sim[diag_range, shifted_diag]
        # InfoNCE loss
        nll = - pos_logits + nn.logsumexp(cos_sim, axis=-1)
```

(continues on next page)

(continued from previous page)

```

nll = nll.mean()

# Logging
metrics = {'loss': nll}
# Determine ranking position of positive example
comb_sim = jnp.concatenate([pos_logits[:,None],
                            cos_sim.at[shifted_diag, diag_range].set(-9e15)],
                           axis=-1)
sim_argsort = (-comb_sim).argsort(axis=-1).argmin(axis=-1)
# Logging of ranking position
metrics['acc_top1'] = (sim_argsort == 0).mean()
metrics['acc_top5'] = (sim_argsort < 5).mean()
metrics['acc_mean_pos'] = (sim_argsort + 1).mean()

return nll, metrics

def encode(self, imgs, train=False):
    # Return features before g()
    model_feats = self.convnet(imgs, train=train)
    return model_feats['block4_1'].mean(axis=(1,2))

```

Alternatively to performing the validation on the contrastive learning loss as well, we could also take a simple, small downstream task, and track the performance of the base network $f(\cdot)$ on that. However, in this tutorial, we will restrict ourselves to the STL10 dataset where we use the task of image classification on STL10 as our test task.

Training

Now that we have implemented SimCLR and the data loading pipeline, we are ready to train the model. We will use the same training function setup as usual. We extend the TrainState with a `batch_stats` attribute for the batch statistics, and an PRNG key `rng` for the augmentations during training.

```
[14]: class TrainState(train_state.TrainState):
    # Batch statistics from BatchNorm
    batch_stats : Any
    # PRNGKey for augmentations
    rng : Any
```

The class `TrainerModule` will build the foundation for our training code and will be specialized for the different parts/models in this tutorial.

```
[15]: class TrainerModule:

    def __init__(self,
                 model_name : str,
                 model_class : Any,
                 eval_key : str,
                 exmp_imgs : Any,
                 lr : float = 1e-3,
                 weight_decay : float = 0.01,
                 seed : int = 42,
                 check_val_every_n_epoch : int = 1,
                 **model_hparams):
```

(continues on next page)

(continued from previous page)

```

"""
Module for summarizing all common training functionalities.

Inputs:
    model_name - Folder name in which to save the checkpoints
    model_class - Module class of the model to train
    eval_key - Name of the metric to check for saving the best model
    exmp_imgs - Example imgs, used as input to initialize the model
    lr - Learning rate of the optimizer to use
    weight_decay - Weight decay to use in the optimizer
    seed - Seed to use in the model initialization
    check_val_every_n_epoch - With which frequency to validate the model
"""

super().__init__()
self.lr = lr
self.weight_decay = weight_decay
self.seed = seed
self.check_val_every_n_epoch = check_val_every_n_epoch
# Create empty model. Note: no parameters yet
self.eval_key = eval_key
self.model_name = model_name
self.model = model_class(**model_hparams)
# Prepare logging
self.log_dir = os.path.join(CHECKPOINT_PATH, f'{self.model_name}/')
self.logger = SummaryWriter(log_dir=self.log_dir)
# Create jitted training and eval functions
self.create_functions()
# Initialize model
self.init_model(exmp_imgs)

def create_functions(self):
    # To be implemented in sub-classes
    raise NotImplementedError

def init_model(self, exmp_imgs):
    # Initialize model
    rng = random.PRNGKey(self.seed)
    rng, init_rng = random.split(rng)
    variables = self.model.init(init_rng, exmp_imgs)
    self.state = TrainState(step=0,
                           apply_fn=self.model.apply,
                           params=variables['params'],
                           batch_stats=variables.get('batch_stats'),
                           rng=rng,
                           tx=None, opt_state=None)

def init_optimizer(self, num_epochs, num_steps_per_epoch):
    # By default, we decrease the learning rate with cosine annealing
    lr_schedule = optax.warmup_cosine_decay_schedule(
        init_value=0.0,
        peak_value=self.lr,
        warmup_steps=0.0,

```

(continues on next page)

(continued from previous page)

```

decay_steps=int(num_epochs * num_steps_per_epoch),
end_value=2e-2*self.lr
)
optimizer = optax.adamw(lr_schedule, weight_decay=self.weight_decay)
self.create_train_state(optimizer)

def create_train_state(self, optimizer):
    # Initialize training state
    self.state = TrainState.create(apply_fn=self.state.apply_fn,
                                    params=self.state.params,
                                    batch_stats=self.state.batch_stats,
                                    rng=self.state.rng,
                                    tx=optimizer)

def train_model(self, train_loader, val_loader, num_epochs=200):
    # Train model for defined number of epochs
    # We first need to create optimizer and the scheduler for the given number of
→epochs
    self.init_optimizer(num_epochs, len(train_loader))
    # Track best eval metric
    best_eval = 0.0
    for epoch_idx in tqdm(range(1, num_epochs+1)):
        self.train_epoch(train_loader, epoch=epoch_idx)
        if epoch_idx % self.check_val_every_n_epoch == 0:
            eval_metrics = self.eval_model(val_loader)
            for key in eval_metrics:
                self.logger.add_scalar(f'val/{key}', eval_metrics[key], global_
→step=epoch_idx)
            if eval_metrics[self.eval_key] >= best_eval:
                best_eval = eval_metrics[self.eval_key]
                self.save_model(step=epoch_idx)
            self.logger.flush()

def train_epoch(self, data_loader, epoch):
    # Train model for one epoch, and log avg metrics
    metrics = defaultdict(float)
    for batch in tqdm(data_loader, desc='Training', leave=False):
        self.state, batch_metrics = self.train_step(self.state, batch)
        for key in batch_metrics:
            metrics[key] += batch_metrics[key]
    num_train_steps = len(data_loader)
    for key in metrics:
        avg_val = metrics[key].item() / num_train_steps
        self.logger.add_scalar('train/'+key, avg_val, global_step=epoch)

def eval_model(self, data_loader):
    # Test model on all images of a data loader and return avg metrics
    metrics = defaultdict(float)
    count = 0
    for batch_idx, batch in enumerate(data_loader):
        batch_metrics = self.eval_step(self.state, random.PRNGKey(batch_idx), batch)
        batch_size = (batch[0] if isinstance(batch, (tuple, list)) else batch).
→shape[0]

```

(continues on next page)

(continued from previous page)

```

    count += batch_size
    for key in batch_metrics:
        metrics[key] += batch_metrics[key] * batch_size
    metrics = {key: metrics[key].item() / count for key in metrics}
    return metrics

def save_model(self, step=0):
    # Save current model at certain training iteration
    checkpoints.save_checkpoint(ckpt_dir=self.log_dir,
                                target={'params': self.state.params,
                                         'batch_stats': self.state.batch_stats},
                                step=step,
                                overwrite=True)

def load_model(self, pretrained=False):
    # Load model. We use different checkpoint for pretrained models
    if not pretrained:
        state_dict = checkpoints.restore_checkpoint(ckpt_dir=self.log_dir,
                                                ↪target=None)
    else:
        state_dict = checkpoints.restore_checkpoint(ckpt_dir=os.path.join(CHECKPOINT_PATH, f'{self.model_name}.ckpt'), target=None)
        num_params = sum([np.prod(p.shape) for p in jax.tree_leaves(state_dict)])
        self.state = TrainState.create(apply_fn=self.state.apply_fn,
                                      params=state_dict['params'],
                                      batch_stats=state_dict['batch_stats'],
                                      rng=self.state.rng,
                                      tx=self.state.tx if self.state.tx else optax.
                                                ↪sgd(self.lr) # Default optimizer
                                      )
    return state_dict

def checkpoint_exists(self):
    # Check whether a pretrained model exist
    return os.path.isfile(os.path.join(CHECKPOINT_PATH, f'{self.model_name}.ckpt'))
```

We specialize this training module for SimCLR below. For saving the best model checkpoint, we track the metric acc_top5, which describes how often the correct image patch is within the top-5 most similar examples in the batch. This is usually less noisy than the top-1 metric, making it a better metric to choose the best model from.

```
[16]: class SimCLRTainer(TrainerModule):

    def __init__(self, **kwargs):
        super().__init__(model_name='SimCLR',
                        model_class=SimCLR,
                        eval_key='acc_top5',
                        **kwargs)

    def create_functions(self):
        # Function to calculate the InfoNCE loss for a batch of images
        def calculate_loss(params, batch_stats, rng, batch, train):
            batch = parallel_augment(rng, batch)
            outs = self.model.apply({'params': params, 'batch_stats': batch_stats},
```

(continues on next page)

(continued from previous page)

```

        batch,
        train=train,
        mutable=['batch_stats'] if train else False)
    (loss, metrics), new_model_state = outs if train else (outs, None)
    return loss, (metrics, new_model_state)

# Training function
def train_step(state, batch):
    rng, forward_rng = random.split(state.rng)
    loss_fn = lambda params: calculate_loss(params,
                                              state.batch_stats,
                                              forward_rng,
                                              batch,
                                              train=True)
    (_, (metrics, new_model_state)), grads = jax.value_and_grad(loss_fn,
                                                               has_
                                                               ↵aux=True)(state.params)
    # Update parameters, batch statistics and PRNG key
    state = state.apply_gradients(grads=grads,
                                   batch_stats=new_model_state['batch_stats'],
                                   rng=rng)
    return state, metrics

# Eval function
def eval_step(state, rng, batch):
    _, (metrics, _) = calculate_loss(state.params,
                                      state.batch_stats,
                                      rng,
                                      batch,
                                      train=False)
    return metrics

# jit for efficiency
self.train_step = jax.jit(train_step)
self.eval_step = jax.jit(eval_step)

```

Finally, let's prepare the data loaders and write a full training function:

```
[17]: def numpy_collate_contrastive(batch):
    imgs1, imgs2 = [[b[0][i] for b in batch] for i in range(2)]
    return np.stack(imgs1 + imgs2, axis=0)

batch_size = 256
simclr_train_loader = data.DataLoader(unlabeled_data,
                                       batch_size=batch_size,
                                       shuffle=True,
                                       drop_last=True,
                                       collate_fn=numpy_collate_contrastive,
                                       num_workers=3,
                                       persistent_workers=True,
                                       generator=torch.Generator().manual_seed(42))
simclr_val_loader = data.DataLoader(train_data_contrast,
                                     batch_size=batch_size,
                                     shuffle=False,
```

(continues on next page)

(continued from previous page)

```
drop_last=False,
collate_fn=numpy_collate_contrastive,
num_workers=3,
persistent_workers=True)
```

A common observation in contrastive learning is that the larger the batch size, the better the models perform. A larger batch size allows us to compare each image to more negative examples, leading to overall smoother loss gradients. However, in our case, we experienced that a batch size of 256 was sufficient to get good results.

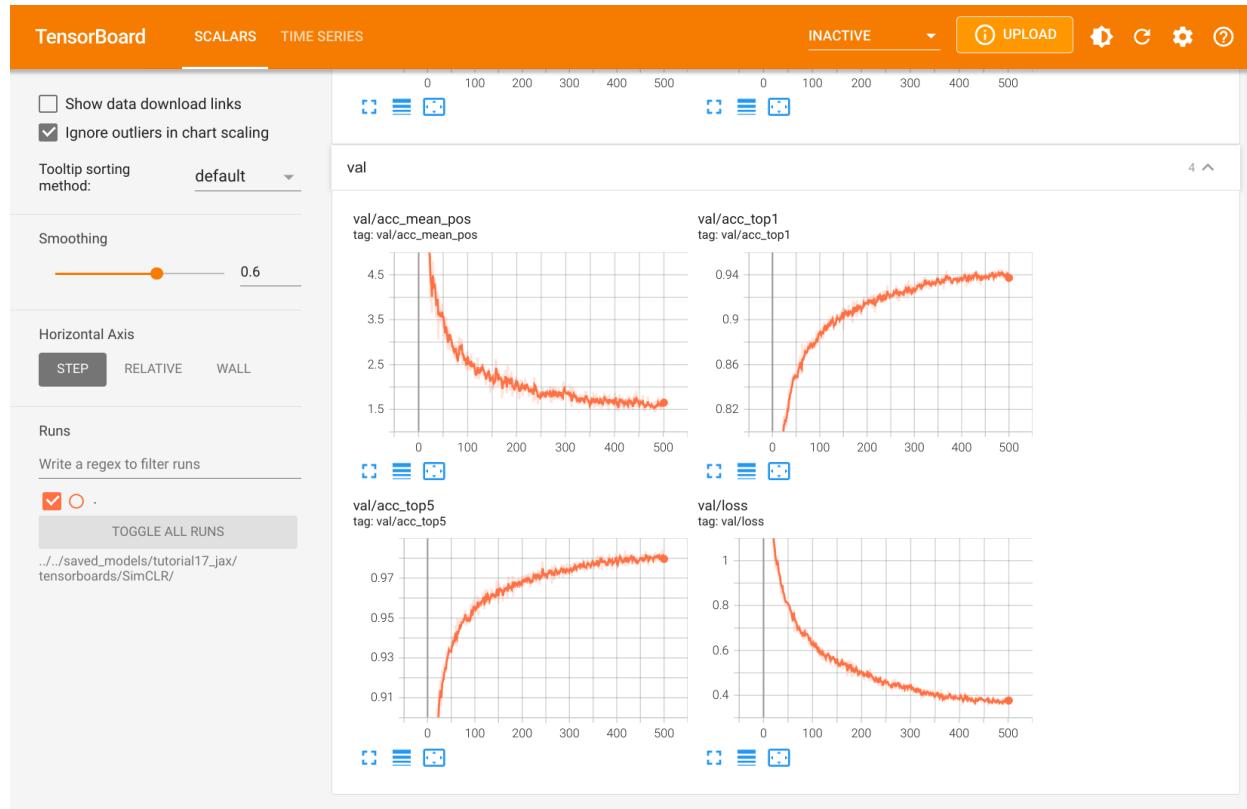
```
[18]: def train_simclr(num_epochs=500, **kwargs):
    # Create a trainer module with specified hyperparameters
    trainer = SimCLRTrainer(exmp_imgs=parallel_augment(random.PRNGKey(0),
                                                       next(iter(simclr_train_loader))),
                           **kwargs)
    if not trainer.checkpoint_exists(): # Skip training if pretrained model exists
        trainer.train_model(simclr_train_loader, simclr_val_loader, num_epochs=num_
        epochs)
        trainer.load_model()
    else:
        trainer.load_model(pretrained=True)
    return trainer
```

With everything ready, let's train the model:

```
[19]: simclr_trainer = train_simclr(hidden_dim=128,
                                   lr=5e-4,
                                   temperature=0.07,
                                   weight_decay=1e-4,
                                   num_epochs=500)
```

To get an intuition of how training with contrastive learning behaves, we can take a look at the TensorBoard below:

```
[20]: %tensorboard --logdir ../../saved_models/tutorial17_jax/tensorboards/SimCLR/
```



One thing to note is that contrastive learning benefits a lot from long training. The shown plot above is from a training that took approx. 16 hrs on a NVIDIA GTX 1080Ti. Training the model for even longer might reduce its loss further, but we did not experience any gains from it for the downstream task on image classification. In general, contrastive learning can also benefit from using larger models, if sufficient unlabeled data is available.

4.29.2 Logistic Regression

After we have trained our model via contrastive learning, we can deploy it on downstream tasks and see how well it performs with little data. A common setup, which also verifies whether the model has learned generalized representations, is to perform Logistic Regression on the features. In other words, we learn a single, linear layer that maps the representations to a class prediction. Since the base network $f(\cdot)$ is not changed during the training process, the model can only perform well if the representations of h describe all features that might be necessary for the task. Further, we do not have to worry too much about overfitting since we have very few parameters that are trained. Hence, we might expect that the model can perform well even with very little data.

First, let's implement a simple Logistic Regression setup for which we assume that the images already have been encoded in their feature vectors. If very little data is available, it might be beneficial to dynamically encode the images during training so that we can also apply data augmentations. However, the way we implement it here is much more efficient and can be trained within a few seconds. Further, using data augmentations did not show any significant gain in this simple setup.

```
[21]: class LogisticRegression(nn.Module):
    num_classes : int

    @nn.compact
    def __call__(self, x):
        return nn.Dense(self.num_classes)(x)
```

```
[22]: class LGTrainer(TrainerModule):

    def __init__(self, model_suffix, **kwargs):
        super().__init__(model_name=f'LogisticRegression_{model_suffix}',
                        model_class=LogisticRegression,
                        eval_key='acc',
                        check_val_every_n_epoch=10,
                        **kwargs)

    def init_optimizer(self, num_epochs, num_steps_per_epoch):
        # We decrease the learning rate with a stepwise function
        lr_schedule = optax.piecewise_constant_schedule(
            init_value=self.lr,
            boundaries_and_scales=
                {int(num_steps_per_epoch * num_epochs * 0.6): 0.1,
                 int(num_steps_per_epoch * num_epochs * 0.8): 0.1}
        )
        optimizer = optax.adamw(lr_schedule, weight_decay=self.weight_decay)
        self.create_train_state(optimizer)

    def create_functions(self):
        # Function to calculate the classification loss and accuracy
        def calculate_loss(params, batch):
            imgs, labels = batch
            logits = self.model.apply({'params': params}, imgs)
            loss = optax.softmax_cross_entropy_with_integer_labels(logits, labels).mean()
            acc = (logits.argmax(axis=-1) == labels).mean()
            metrics = {'loss': loss, 'acc': acc}
            return loss, metrics

        # Training function
        def train_step(state, batch):
            _, metrics, grads = jax.value_and_grad(calculate_loss,
                                                    has_aux=True)(state.params, batch)
            state = state.apply_gradients(grads=grads)
            return state, metrics

        # Eval function
        def eval_step(state, rng, batch):
            _, metrics = calculate_loss(state.params, batch)
            return metrics

        # jit for efficiency
        self.train_step = jax.jit(train_step)
        self.eval_step = jax.jit(eval_step)


```

The data we use is the training and test set of STL10. The training contains 500 images per class, while the test set has 800 images per class.

```
[23]: def numpy_collate(batch):
    if isinstance(batch[0], np.ndarray):
        return np.stack(batch)
    elif isinstance(batch[0], (tuple, list)):
        transposed = zip(*batch)
        return [numpy_collate(samples) for samples in transposed]
    else:
```

(continues on next page)

(continued from previous page)

```

    return np.array(batch)

img_transforms = transforms.Compose([image_to_numpy,
                                    lambda img: (img * 2.0 - 1.0)])

train_img_data = STL10(root=DATASET_PATH, split='train', download=True,
                      transform=img_transforms)
test_img_data = STL10(root=DATASET_PATH, split='test', download=True,
                      transform=img_transforms)

print("Number of training examples:", len(train_img_data))
print("Number of test examples:", len(test_img_data))

Files already downloaded and verified
Files already downloaded and verified
Number of training examples: 5000
Number of test examples: 8000

```

Next, we implement a small function to encode all images in our datasets. The output representations are then used as inputs to the Logistic Regression model.

```
[24]: class NumpyDataset(data.Dataset):
    # data.TensorDataset for numpy arrays

    def __init__(self, *arrays):
        self.arrays = arrays

    def __len__(self):
        return self.arrays[0].shape[0]

    def __getitem__(self, idx):
        return [arr[idx] for arr in self.arrays]
```

```
[25]: def prepare_data_features(encode_fn, dataset):
    # Encode all images
    data_loader = data.DataLoader(dataset, batch_size=64,
                                  shuffle=False,
                                  drop_last=False,
                                  collate_fn=numpy_collate)
    feats, labels = [], []
    for batch_imgs, batch_labels in tqdm(data_loader):
        batch_feats = encode_fn(batch_imgs)
        feats.append(jax.device_get(batch_feats))
        labels.append(batch_labels)

    feats = np.concatenate(feats, axis=0)
    labels = np.concatenate(labels, axis=0)

    # Sort images by labels for easier postprocessing later
    idxs = labels.argsort()
    labels, feats = labels[idxs], feats[idxs]
```

(continues on next page)

(continued from previous page)

```
    return NumpyDataset(feats, labels)
```

Let's apply the function to both training and test set below.

```
[26]: simclr_model = simclr_trainer.model.bind({'params': simclr_trainer.state.params,
                                              'batch_stats': simclr_trainer.state.batch_
                                              ↪stats})
encode_fn = jax.jit(lambda img: simclr_model.encode(img))
train_feats_simclr = prepare_data_features(encode_fn, train_img_data)
test_feats_simclr = prepare_data_features(encode_fn, test_img_data)

0%|          | 0/79 [00:00<?, ?it/s]
0%|          | 0/125 [00:00<?, ?it/s]
```

Finally, we can write a training function as usual. We evaluate the model on the test set every 10 epochs to allow early stopping, but the low frequency of the validation ensures that we do not overfit too much on the test set.

```
[27]: def train_logreg(batch_size, train_feats_data, test_feats_data, model_suffix, num_
                    ↪epochs=100, **kwargs):
    # Data loaders
    train_loader = data.DataLoader(train_feats_data,
                                    batch_size=batch_size,
                                    shuffle=True,
                                    drop_last=False,
                                    generator=torch.Generator().manual_seed(42),
                                    collate_fn= numpy_collate)
    test_loader = data.DataLoader(test_feats_data,
                                 batch_size=batch_size,
                                 shuffle=False,
                                 drop_last=False,
                                 collate_fn=numpy_collate)

    # Create a trainer module with specified hyperparameters
    trainer = LGTrainer(exmp_imgs=next(iter(train_loader))[0],
                         model_suffix=model_suffix,
                         **kwargs)
    if not trainer.checkpoint_exists(): # Skip training if pretrained model exists
        trainer.train_model(train_loader, test_loader, num_epochs=num_epochs)
        trainer.load_model()
    else:
        trainer.load_model(pretrained=True)

    # Test best model on train and validation set
    train_result = trainer.eval_model(train_loader)
    test_result = trainer.eval_model(test_loader)
    result = {"train": train_result["acc"], "test": test_result["acc"]}

    return trainer, result
```

Despite the training dataset of STL10 already only having 500 labeled images per class, we will perform experiments with even smaller datasets. Specifically, we train a Logistic Regression model for datasets with only 10, 20, 50, 100, 200, and all 500 examples per class. This gives us an intuition on how well the representations learned by contrastive

learning can be transferred to a image recognition task like this classification. First, let's define a function to create the intended sub-datasets from the full training set:

```
[28]: def get_smaller_dataset(original_dataset, num_imgs_per_label):
    new_arrays = [t.reshape(10, -1, *t.shape[1:])[ :, :num_imgs_per_label].reshape(-1, *t.
    ↵shape[1:])]
        for t in original_dataset.arrays]
    new_dataset = NumpyDataset(*new_arrays)
    return new_dataset
```

Next, let's run all models. Despite us training 6 models, this cell could be run within a minute or two without the pretrained models.

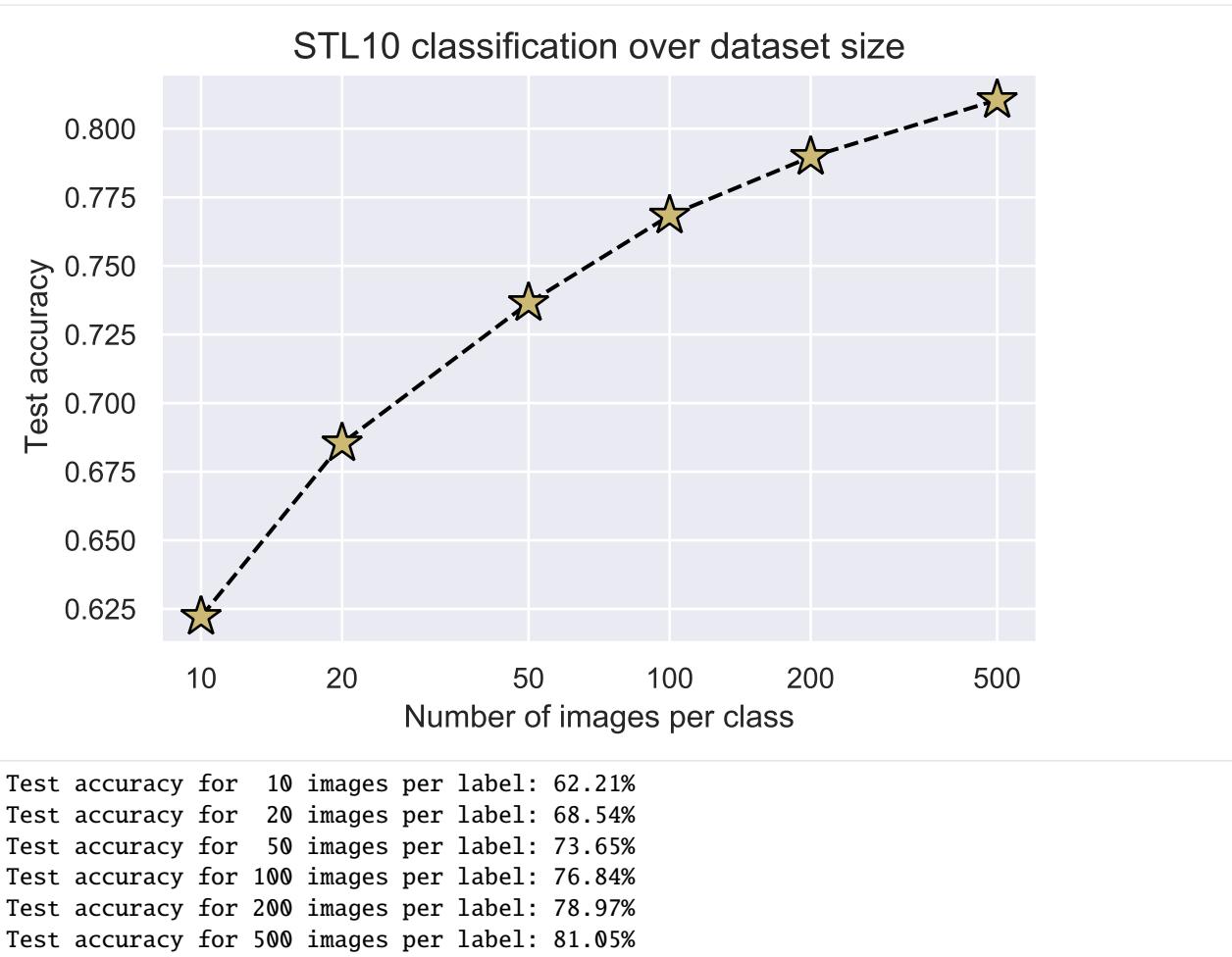
```
[29]: results = {}
for num_imgs_per_label in [10, 20, 50, 100, 200, 500]:
    sub_train_set = get_smaller_dataset(train_feats_simclr, num_imgs_per_label)
    _, small_set_results = train_logreg(batch_size=64,
                                         train_feats_data=sub_train_set,
                                         test_feats_data=test_feats_simclr,
                                         model_suffix=num_imgs_per_label,
                                         num_classes=10,
                                         lr=1e-3,
                                         weight_decay=1e-3)
    results[num_imgs_per_label] = small_set_results
```

Finally, let's plot the results.

```
[30]: dataset_sizes = sorted([k for k in results])
test_scores = [results[k]["test"] for k in dataset_sizes]

fig = plt.figure(figsize=(6,4))
plt.plot(dataset_sizes, test_scores, '--', color="#000", marker="*", markeredgecolor=""
         ↵#000", markerfacecolor="y", markersize=16)
plt.xscale("log")
plt.xticks(dataset_sizes, labels=dataset_sizes)
plt.title("STL10 classification over dataset size", fontsize=14)
plt.xlabel("Number of images per class")
plt.ylabel("Test accuracy")
plt.minorticks_off()
plt.show()

for k, score in zip(dataset_sizes, test_scores):
    print(f'Test accuracy for {k:3d} images per label: {100*score:.2f}%')
```



As one would expect, the classification performance improves the more data we have. However, with only 10 images per class, we can already classify more than 60% of the images correctly. This is quite impressive, considering that the images are also higher dimensional than e.g. CIFAR10. With the full dataset, we achieve an accuracy of 81%. The increase between 50 to 500 images per class might suggest a linear increase in performance with an exponentially larger dataset. However, with even more data, we could also finetune $f(\cdot)$ in the training process, allowing for the representations to adapt more to the specific classification task given.

To set the results above into perspective, we will train the base network, a ResNet-18, on the classification task from scratch.

4.29.3 Baseline

As a baseline to our results above, we will train a standard ResNet-18 with random initialization on the labeled training set of STL10. The results will give us an indication of the advantages that contrastive learning on unlabeled data has compared to using only supervised training. The implementation of the model is straightforward since the ResNet architecture is provided in the flaxmodels library.

```
[31]: class ResNet(nn.Module):
    num_classes : int

    @nn.compact
```

(continues on next page)

(continued from previous page)

```
def __call__(self, x, train=True):
    return flaxmodels.ResNet18(output='logits',
                               pretrained=False,
                               normalize=False,
                               num_classes=self.num_classes)(x, train=train)
```

It is clear that the ResNet easily overfits on the training data since its parameter count is more than 1000 times larger than the dataset size. To make the comparison to the contrastive learning models fair, we apply data augmentations similar to the ones we used before: horizontal flip, crop-and-resize, grayscale, and gaussian blur. Color distortions as before are not used because the color distribution of an image showed to be an important feature for the classification. Hence, we observed no noticeable performance gains when adding color distortions to the set of augmentations. Similarly, we restrict the resizing operation before cropping to the max. 125% of its original resolution, instead of 1250% as done in SimCLR. This is because, for classification, the model needs to recognize the full object, while in contrastive learning, we only want to check whether two patches belong to the same image/object. Hence, the chosen augmentations below are overall weaker than in the contrastive learning case.

Following the same setup as for SimCLR, we apply the resizing augmentation in the data loading, and the remaining augmentations efficiently on GPU.

```
[32]: train_transforms = transforms.Compose([transforms.RandomResizedCrop(size=96, scale=(0.8, 1.0)),
                                          image_to_numpy])

train_img_aug_data = STL10(root=DATASET_PATH, split='train', download=True,
                           transform=train_transforms)

Files already downloaded and verified
```

```
[33]: def augment_image_for_classification(rng, img):
    rngs = random.split(rng, 3)
    # Random left-right flip
    img = dm_pix.random_flip_left_right(rngs[0], img)
    # Random grayscale
    should_gs = random.bernoulli(rngs[1], p=0.2)
    img = jax.lax.cond(should_gs, # Only apply grayscale if true
                       lambda x: dm_pix.rgb_to_grayscale(x, keep_dims=True),
                       lambda x: x,
                       img)
    # Gaussian blur
    sigma = random.uniform(rngs[2], shape=(1,), minval=0.1, maxval=0.5)
    img = dm_pix.gaussian_blur(img, sigma=sigma[0], kernel_size=9)
    # Normalization
    img = img * 2.0 - 1.0
    return img

resnet_augm = jax.jit(lambda rng, imgs: jax.vmap(augment_image_for_
                                                 classification)(random.split(rng, imgs.shape[0])), imgs))
```

Using this augmentation function, we can define our trainer module, which is an extension of the Logistic Regression trainer to tracking BatchNorm statistics and augmentations.

```
[34]: class RNTrainer(TrainerModule):
```

(continues on next page)

(continued from previous page)

```

def __init__(self, **kwargs):
    super().__init__(model_name=f'ResNet',
                    model_class=ResNet,
                    eval_key='acc',
                    check_val_every_n_epoch=2,
                    **kwargs)

def init_optimizer(self, num_epochs, num_steps_per_epoch):
    # We decrease the learning rate with a stepwise function
    lr_schedule = optax.piecewise_constant_schedule(
        init_value=self.lr,
        boundaries_and_scales=
            {int(num_steps_per_epoch * num_epochs * 0.7): 0.1,
             int(num_steps_per_epoch * num_epochs * 0.9): 0.1}
    )
    optimizer = optax.adamw(lr_schedule, weight_decay=self.weight_decay)
    self.create_train_state(optimizer)

def create_functions(self):
    # Function to calculate the classification loss and accuracy
    def calculate_loss(params, batch_stats, rng, batch, train):
        imgs, labels = batch
        if train:
            imgs = resnet_augm(rng, imgs)
        outs = self.model.apply({'params': params, 'batch_stats': batch_stats},
                               imgs,
                               train=train,
                               mutable=['batch_stats'] if train else False)
        logits, new_model_state = outs if train else (outs, None)
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, labels).mean()
        acc = (logits.argmax(axis=-1) == labels).mean()
        metrics = {'loss': loss, 'acc': acc}
        return loss, (metrics, new_model_state)
    # Training function
    def train_step(state, batch):
        rng, forward_rng = random.split(state.rng)
        loss_fn = lambda params: calculate_loss(params,
                                                state.batch_stats,
                                                forward_rng,
                                                batch,
                                                train=True)
        # Get loss, gradients for loss, and other outputs of loss function
        _, (metrics, new_model_state), grads = jax.value_and_grad(loss_fn,
                                                               has_
                                                               ↵aux=True)(state.params)
        # Update parameters, batch statistics and PRNG key
        state = state.apply_gradients(grads=grads,
                                      batch_stats=new_model_state['batch_stats'],
                                      rng=rng)
        return state, metrics
    # Eval function
    def eval_step(state, rng, batch):

```

(continues on next page)

(continued from previous page)

```

    _, (metrics, _) = calculate_loss(state.params,
                                      state.batch_stats,
                                      rng,
                                      batch,
                                      train=False)

    return metrics
# jit for efficiency
self.train_step = jax.jit(train_step)
self.eval_step = jax.jit(eval_step)

```

The training function for the ResNet is almost identical to the Logistic Regression setup. Note that we allow the ResNet to perform validation every 2 epochs to also check whether the model overfits strongly in the first iterations or not.

```
[35]: def train_resnet(batch_size, num_epochs=100, **kwargs):
    # Data loaders
    train_loader = data.DataLoader(train_img_aug_data,
                                    batch_size=batch_size,
                                    shuffle=True,
                                    drop_last=False,
                                    generator=torch.Generator().manual_seed(42),
                                    num_workers=3,
                                    persistent_workers=True,
                                    collate_fn= numpy_collate)
    test_loader = data.DataLoader(test_img_data,
                                  batch_size=batch_size,
                                  shuffle=False,
                                  drop_last=False,
                                  num_workers=3,
                                  persistent_workers=True,
                                  collate_fn=numpy_collate)

    # Create a trainer module with specified hyperparameters
    trainer = RNTrainer(exmp_imgs=next(iter(train_loader))[0],
                         **kwargs)
    if not trainer.checkpoint_exists(): # Skip training if pretrained model exists
        trainer.train_model(train_loader, test_loader, num_epochs=num_epochs)
        trainer.load_model()
    else:
        trainer.load_model(pretrained=True)

    # Test best model on validation/test set
    test_result = trainer.eval_model(test_loader)
    result = {"test": test_result["acc"]}

    return trainer, result
```

Finally, let's train the model and check its results:

```
[36]: resnet_model, resnet_result = train_resnet(batch_size=64,
                                              num_classes=10,
                                              lr=1e-3,
                                              weight_decay=2e-4,
```

(continues on next page)

(continued from previous page)

```
        num_epochs=100)
print(f"Accuracy on test set: {resnet_result['test']:4.2%}")

Accuracy on test set: 71.64%
```

The ResNet trained from scratch achieves 71.64% on the test set. This is almost 8% less than the contrastive learning model, and even slightly less than SimCLR achieves with 1/10 of the data. This shows that self-supervised, contrastive learning provides considerable performance gains by leveraging large amounts of unlabeled data when little labeled data is available.

4.29.4 Conclusion

In this tutorial, we have discussed self-supervised contrastive learning and implemented SimCLR as an example method. We have applied it to the STL10 dataset and showed that it can learn generalizable representations that we can use to train simple classification models. With 500 images per label, it achieved an 8% higher accuracy than a similar model solely trained from supervision and performs on par with it when only using a tenth of the labeled data. Our experimental results are limited to a single dataset, but recent works such as Ting Chen et al. showed similar trends for larger datasets like ImageNet. Besides the discussed hyperparameters, the size of the model seems to be important in contrastive learning as well. If a lot of unlabeled data is available, larger models can achieve much stronger results and come close to their supervised baselines. Further, there are also approaches for combining contrastive and supervised learning, leading to performance gains beyond supervision (see Khosla et al.). Moreover, contrastive learning is not the only approach to self-supervised learning that has come up in the last two years and showed great results. Other methods include distillation-based methods like BYOL and redundancy reduction techniques like Barlow Twins. There is a lot more to explore in the self-supervised domain, and more, impressive steps ahead are to be expected.

References

- [1] Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. (2020). A simple framework for contrastive learning of visual representations. In International Conference on Machine Learning (ICML 2020). PMLR. ([link](#))
- [2] Chen, T., Kornblith, S., Swersky, K., Norouzi, M., and Hinton, G. (2020). Big self-supervised models are strong semi-supervised learners. NeurIPS 2021 ([link](#)).
- [3] Oord, A. V. D., Li, Y., and Vinyals, O. (2018). Representation learning with contrastive predictive coding. arXiv preprint arXiv:1807.03748. ([link](#))
- [4] Grill, J.B., Strub, F., Altché, F., Tallec, C., Richemond, P.H., Buchatskaya, E., Doersch, C., Pires, B.A., Guo, Z.D., Azar, M.G. and Piot, B. (2020). Bootstrap your own latent: A new approach to self-supervised learning. NeurIPS 2020 ([link](#))
- [5] Khosla, P., Teterwak, P., Wang, C., Sarna, A., Tian, Y., Isola, P., Maschinot, A., Liu, C. and Krishnan, D. (2020). Supervised contrastive learning. NeurIPS 2020 ([link](#))
- [6] Zbontar, J., Jing, L., Misra, I., LeCun, Y. and Deny, S. (2021). Barlow twins: Self-supervised learning via redundancy reduction. In International Conference on Machine Learning (ICML 2021). ([link](#))

If you found this tutorial helpful, consider -ing our repository.

For any questions, typos, or bugs that you found, please raise an issue on GitHub.

4.30 GDL - Regular Group Convolutions

Filled notebook:

Pre-trained models:

Authors: David Knigge, Gabriele Cesa

4.30.1 0. Introduction

In this notebook, we will be implementing regular group convolutional networks from scratch, only making use of pytorch primitives. The goal is to get familiar with the practical considerations to take into account when actually implementing these convolutional networks. Questions and feedback may be forwarded to David Knigge; d.m.knigge@uva.nl.

If you'd like a refresher of the lecture, here we give a brief overview of the operations we are going to work with / implement. These will be treated more extensively below. For simplicity of notation, here we assume each CNN layer consists of only a single channel.

0.1 Brief recap on CNNs

Conventional CNNs make use of the convolution operator, here defined over \mathbb{R}^2 for a signal $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ and a kernel $k : \mathbb{R}^2 \rightarrow \mathbb{R}$ at $\mathbf{x} \in \mathbb{R}^2$:

$$(f * k)(\mathbf{x}) = \int_{\mathbb{R}^2} f(\tilde{\mathbf{x}})k(\tilde{\mathbf{x}} - \mathbf{x})d\tilde{\mathbf{x}},$$

As we can see, the convolution operation comes down to an inner product of the function f and a shifted kernel k .

Sidenote: In reality CNNs implement a discretised version of this operation;

$$\begin{aligned}(f * k)(\mathbf{x}) &= \sum_{\tilde{\mathbf{x}} \in \mathbb{Z}^2} f(\tilde{\mathbf{x}})k(\mathbf{x} - \tilde{\mathbf{x}})\Delta\tilde{\mathbf{x}} \\ &= \sum_{\tilde{\mathbf{x}} \in \mathbb{Z}^2} f(\tilde{\mathbf{x}})k(\mathbf{x} - \tilde{\mathbf{x}})\end{aligned}$$

Where above, since pixels in an image are generally evenly spaced, we set $\Delta\tilde{\mathbf{x}} = 1$. For this recap we stay in the continuous domain for simplicity.

In convolution layers, like PyTorch's Conv2D implementation, the above operation is carried out for every $\mathbf{x} \in \mathbb{Z}^2$ (limited of course to the domain over which the image is defined). Because the same set of weights is used throughout the input, the output of this operation is [equivariant](#) to transformations from the translation group \mathbb{R}^2 . Furthermore f, k usually consist of a number of channels, which are all summed over.

In this tutorial, we will use PyTorch's `torch.nn.functional.conv2d()` function to perform this integration operation at every position in input feature map. This saves us having to implement the convolution operation ourselves.

0.2 Brief recap on GCNNs

In regular group convolutions, the goal is to have a CNN of which are not only equivariant to translations \mathbb{R}^2 , but which are also equivariant to another (usually broader) group of interest G . We focus specifically on groups which are combinations of translations \mathbb{R}^2 and some group of interest H . In this tutorial we keep to the group of 90 degree rotations in 2D; the [Cyclic group](#) of order 4 $H = C_4$.

We will operate on 2D images, generally defined on \mathbb{R}^2 , as such, the first step in constructing a network which can track under which *pose* (read: transformation from a group $G = \mathbb{R}^2 \rtimes H$) a feature in the input occurs, we need to transfer our signal to a domain in which the same feature under a different pose is disentangled. This happens through *the lifting convolution*, which maps features in our input signal $f_{in} : \mathbb{R}^2 \rightarrow \mathbb{R}$ to a feature map on the group $f_{out} : G \rightarrow \mathbb{R}$. For a signal and kernel f, k both defined on \mathbb{R}^2 , and a group element $g = (\mathbf{x}, h) \in G = \mathbb{R}^2 \rtimes H$:

$$(f * \text{lifting } k)(g) = \int_{\mathbb{R}^2} f(\tilde{\mathbf{x}}) k_h(\tilde{\mathbf{x}} - \mathbf{x}) d\tilde{\mathbf{x}}.$$

Where k_h is the kernel $k : \mathbb{R}^2 \rightarrow \mathbb{R}$ transformed under the regular representation \mathcal{L}_h of a group element $h \in H$; $k_h = \frac{1}{|h|} \mathcal{L}_h[k]$.

Sidenote: The factor $\frac{1}{|h|}$, with $|h|$ the determinant of the matrix representation of h in \mathbb{R}^2 , accounts for a possible change in volume on \mathbb{R}^2 that h might have. Working with the cyclic group, we don't encounter this problem (the determinant of a rotation matrix is 1, volumes are invariant to rotations on \mathbb{R}^2), but if you'd like to implement equivariance to for example the dilation group, this becomes important.

Next, now that we have a feature map defined on the group; $f_{out} : G \rightarrow \mathbb{R}$, we apply group convolutions, extending the convolution operation to an integral over the entire group G ;

$$\begin{aligned} (f * \text{group } k)(g) &= \int_G f(\tilde{g}) k(g^{-1} \cdot \tilde{g}) d\tilde{g} \\ &= \int_{\mathbb{R}^2} \int_H f(\tilde{\mathbf{x}}, \tilde{h}) \mathcal{L}_x \mathcal{L}_h k(\tilde{\mathbf{x}}, \tilde{h}) \frac{1}{|h|} d\tilde{\mathbf{x}} d\tilde{h} \\ &= \int_{\mathbb{R}^2} \int_H f(\tilde{\mathbf{x}}, \tilde{h}) k(h^{-1}(\tilde{\mathbf{x}} - \mathbf{x}), h^{-1} \cdot \tilde{h}) \frac{1}{|h|} d\tilde{\mathbf{x}} d\tilde{h}. \end{aligned}$$

The main difference with the lifting convolution is that the signal and kernel f, k are both functions on $G; G \rightarrow \mathbb{R}$, and the integral reflects this by extending over the entire group G . Other than that, there is little difference!

After a number of such group convolutional layers, we will want to ultimately obtain a representation that is invariant to the group action. We can do this by performing a projection which collapses our function defined over G to a single point, with an operation that is invariant to the group action (summing, averaging, max, min).

After this short refresher, let's get to coding!

0.3 Installing and importing some useful packages

Here we install and import some libraries that we will use throughout this tutorial. We use the [pytorch](#) as our deep learning framework of choice. Note that for ease of model training and tracking, we additionally make use of [pytorch-lightning](#). Although the specifics of this package aren't the focus of this tutorial, if you'd like a refresher please have a look at this excellent [Deep Learning 1 tutorial](#) by Phillip Lippe.

```
[1]: ## Standard libraries
import os
import numpy as np
import math
from PIL import Image
```

(continues on next page)

(continued from previous page)

```

from types import SimpleNamespace
from functools import partial

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0

## PyTorch
import torch
import torch.nn as nn
import torch.utils.data as data
import torch.optim as optim
## Torchvision
import torchvision
from torchvision.datasets import MNIST
from torchvision import transforms
## PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip3 install pytorch-lightning>=1.4 --quiet
    import pytorch_lightning as pl
import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

```

[2]: # Path to the folder where the datasets are be downloaded (e.g. MNIST)
 DATASET_PATH = ".../data"
 # Path to the folder where the pretrained models are saved
 CHECKPOINT_PATH = ".../saved_models/DL2/GDL"
 # Ensure that all operations are deterministic on GPU (if used) for reproducibility
 torch.backends.cudnn.deterministic = True
 torch.backends.cudnn.benchmark = False
 device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")

[3]: import urllib.request
 from urllib.error import HTTPError
 # Github URL where saved models are stored for this tutorial
 base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/DL2/GDL/"
 # Files to download
 pretrained_files = ["cnn-pretrained.ckpt", "gcnn-pretrained.ckpt", "paprika.tiff"]
 # Create checkpoint path if it doesn't exist yet
 os.makedirs(CHECKPOINT_PATH, exist_ok=True)

(continues on next page)

(continued from previous page)

```
# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive_"
            "folder, or contact the author with the full output including the following error:\n", e)
```

4.30.2 1. Group theory

1.1 What is a group?

To start off, we recap some of the group theoretical preliminaries that we saw in the first lecture of this module. Recall that a group is defined by a tuple (G, \cdot) , where G is a set of group elements and \cdot the binary group action which tells us how elements $g \in G$ combine. The group action \cdot needs to satisfy:

1. Closure. G is closed under \cdot ; for all $g_1, g_2 \in G$ we have $g_1 \cdot g_2 \in G$.
2. Identity. There exists an identity element e s.t. for each $g \in G$, we have $e \cdot g = g \cdot e = g$.
3. Inverse. For every element $g \in G$ we have an element $g^{-1} \in G$, s.t. $g \cdot g^{-1} = e$.
4. Associativity. For every set of elements $g_1, g_2, g_3 \in G$, we have $(g_1 \cdot g_2) \cdot g_3 = g_1 \cdot (g_2 \cdot g_3)$.

The group can have an action on functions defined on \mathbb{R}^2 , which we can instantiate through the *regular representation* $\mathcal{L}_g^{\mathbb{G} \rightarrow \mathbb{R}^2}$. For simplicity, we write \mathcal{L}_g . It is given by:

$$\mathcal{L}_g f(\mathbf{x}) = f(g^{-1} \cdot \mathbf{x})$$

Where we write the action of g^{-1} on x as $g^{-1} \cdot \mathbf{x}$. This is where the regular group convolution gets its name; because of its use of the regular representation to transform the kernels k used throughout the network.

1.2 Implementing a group in python

Let's start out with a baseclass in which we outline which functions we are going to need when working with groups in our setting. As we're going to use `torch` in our implementation of group convolutional neural networks, let us implement the group as a `torch` module as well.

We first specify a base class `GroupBase` in which we specify all necessary properties and operations that we need in our treatment of group convolutional neural networks. The idea is that in our implementation of group convolutions, implementing these functions is necessary and sufficient for extending group convolutional neural networks to other groups. In other words; if you'd like to implement group convolutions equivariant to a new group you find interesting, just inherit this baseclass, implement its methods, and you're good to go. (In practice this only faithfully works for discrete, compact groups).

```
[4]: class GroupBase(torch.nn.Module):

    def __init__(self, dimension, identity):
```

(continues on next page)

(continued from previous page)

```

    """ Implements a group.

    @param dimension: Dimensionality of the group (number of dimensions in the basis
        ↪ of the algebra).
    @param identity: Identity element of the group.
    """
    super().__init__()
    self.dimension = dimension
    self.register_buffer('identity', torch.Tensor(identity))

    def elements(self):
        """ Obtain a tensor containing all group elements in this group.

        """
        raise NotImplementedError()

    def product(self, h, h_prime):
        """ Defines group product on two group elements.

        @param g1: Group element 1
        @param g2: Group element 2
        """
        raise NotImplementedError()

    def inverse(self, h):
        """ Defines inverse for group element.

        @param g: A group element.
        """
        raise NotImplementedError()

    def left_action_on_R2(self, h_batch, x_batch):
        """ Group action of an element from the subgroup H on a vector in R2. For
            ↪ efficiency we
            implement this batchwise.

        @param h_batch: Group elements from H.
        @param x_batch: Vectors in R2.
        """
        raise NotImplementedError()

    def left_action_on_H(self, h_batch, h_prime_batch):
        """ Group action of elements of H on other elements in H itself. Comes down to
            ↪ group product.
            For efficiency we implement this batchwise. Each element in h_batch is applied
            ↪ to each element
            in h_prime_batch.

        @param h_batch: Group elements from H.
        @param h_prime_batch: Other group elements in H.
        """
        raise NotImplementedError()

```

(continues on next page)

(continued from previous page)

```

def matrix_representation(self, h):
    """ Obtain a matrix representation in R^2 for an element h.

    @param h: Group element
    """
    raise NotImplementedError()

def determinant(self, h):
    """ Calculate the determinant of the representation of a group element
    h.

    @param g:
    """
    raise NotImplementedError()

def normalize_group_elements(self, h):
    """ Map the group elements to an interval [-1, 1]. We use this to create
    a standardized input for obtaining weights over the group.

    @param g:
    """
    raise NotImplementedError()

```

1.3 Implementing the cyclic group C_4

As an example, let's discuss a relatively simple group; the group of all 90 rotations of the plane, otherwise known as the cyclic group C_4 . Note:

- The set of group elements of C_4 is given by $G := \{e, g, g^2, g^3\}$. We can parameterise these group elements using rotation angles θ , i.e. $e = 0, g = \frac{1}{2}\pi, g^2 = \pi, \dots$
- The group product is then given by $g \cdot g' := \theta + \theta' \bmod 2\pi$.
- The inverse is given by: $g^{-1} = -\theta \bmod 2\pi$.
- The group C_4 has an action on the euclidean plane in 2 dimensions \mathbb{R}^2 given by a rotation matrix;

$$R_\theta : \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

This gives us the regular representation \mathcal{L}_θ on functions f defined over \mathbb{R}^2 :

$$\mathcal{L}_\theta f(\mathbf{x}) = f(R_{-\theta} \bmod 2\pi \mathbf{x}).$$

Let's implement this group!

```
[5]: class CyclicGroup(GroupBase):

    def __init__(self, order):
        super().__init__(
            dimension=1,
            identity=[0.])

```

(continues on next page)

(continued from previous page)

```

)
    assert order > 1
    self.order = torch.tensor(order)

def elements(self):
    """ Obtain a tensor containing all group elements in this group.

    """
    return torch.linspace(
        start=0,
        end=2 * np.pi * float(self.order - 1) / float(self.order),
        steps=self.order,
        device=self.identity.device
    )

def product(self, h1, h2):
    """ Defines group product on two group elements of the cyclic group C4.

    @param h1: Group element 1
    @param h2: Group element 2
    """

    # As we directly parameterize the group by its rotation angles
    return torch.remainder(h1 + h2, 2 * np.pi)

def inverse(self, h):
    """ Defines group inverse for an element of the cyclic group C4.

    @param h: Group element
    """

    return torch.remainder(-h, 2 * np.pi)

def left_action_on_R2(self, batch_h, batch_x):
    """ Group action of an element g on a set of vectors in R2.

    @param batch_h: Tensor of group elements.
    @param batch_x: Tensor of vectors in R2.
    """

    # Create a tensor containing representations of each of the group
    # elements in the input. Creates a tensor of size [batch_size, 2, 2].
    batched_rep = torch.stack([self.matrix_representation(h) for h in batch_h])

    # Transform the r2 input grid with each representation to end up with
    # a transformed grid of dimensionality [num_group_elements, spatial_dim_y,
    # spatial_dim_x, 2].
    out = torch.einsum('boi,ixy->bxyo', batched_rep, batch_x)

    # Afterwards (because grid_sample assumes our grid is y,x instead of x,y)
    # we swap x and y coordinate values with a roll along final dimension.
    return out.roll(shifts=1, dims=-1)

```

(continues on next page)

(continued from previous page)

```

def left_action_on_H(self, batch_h, batch_h_prime):
    """ Group action of an element h on a set of group elements in H.
    Nothing more than a batchwise group product.

    @param batch_h: Tensor of group elements.
    @param batch_h_prime: Tensor of group elements to apply group product to.
    """
    # The elements in batch_h work on the elements in batch_h_prime directly,
    # through the group product. Each element in batch_h is applied to each element
    # in batch_h_prime.
    transformed_batch_h = self.product(batch_h.repeat(batch_h_prime.shape[0], 1),
                                         batch_h_prime.unsqueeze(-1))
    return transformed_batch_h

def matrix_representation(self, h):
    """ Obtain a matrix representation in R^2 for an element h.

    @param h: A group element.
    """
    cos_t = torch.cos(h)
    sin_t = torch.sin(h)

    return torch.tensor([
        [cos_t, -sin_t],
        [sin_t, cos_t]
    ], device=self.identity.device)

def normalize_group_elements(self, h):
    """ Normalize values of group elements to range between -1 and 1.
    The group elements range from 0 to 2pi * (self.order - 1) / self.order,
    so we normalize by

    @param h: A group element.
    :return:
    """
    largest_elem = 2 * np.pi * (self.order - 1) / self.order

    return (2*h / largest_elem) - 1.

```

1.4 Visualizing the group action

Let's play around with the group implementation we have just created!

```
[6]: img = Image.open(os.path.join(CHECKPOINT_PATH, "paprika.tiff"))

img_tensor = transforms.ToTensor()(img)

img
```

[6]:



[7]: # This creates a grid of the pixel locations in our image

```
img_grid_R2 = torch.stack(torch.meshgrid(  
    torch.linspace(-1, 1, img_tensor.shape[-1]),  
    torch.linspace(-1, 1, img_tensor.shape[-2]),  
))
```

```
# [2, 512, 512] since our image is 2 dimensional and has a width and height of  
# 512 pixels  
img_grid_R2.shape
```

[7]: `torch.Size([2, 512, 512])`

[8]: # let's create the group of 90 degree clockwise rotations

```
c4 = CyclicGroup(order=4)  
e, g1, g2, g3 = c4.elements()
```

[9]: # What happens when we take the product of two rotations?

```
g = c4.product(g1, g2)
```

(continues on next page)

(continued from previous page)

```
# Transform the image grid we just created with the matrix representation of
# this group element. Note that we implemented this batchwise, so we add a dim.
transformed_grid = c4.left_action_on_R2(c4.inverse(g).unsqueeze(0), img_grid_R2)
```

```
[10]: # This function samples an input tensor based on a grid using interpolation.
# It is implemented for batchwise operations so we add a dimension to our and input_
# image.
transformed_img = torch.nn.functional.grid_sample(img_tensor.unsqueeze(0), transformed_
grid, align_corners=True, mode="bilinear")

# if we turn this back into a PIL image we can see the result of our transformation!
transforms.ToPILImage()(transformed_img[0])
```

[10]:

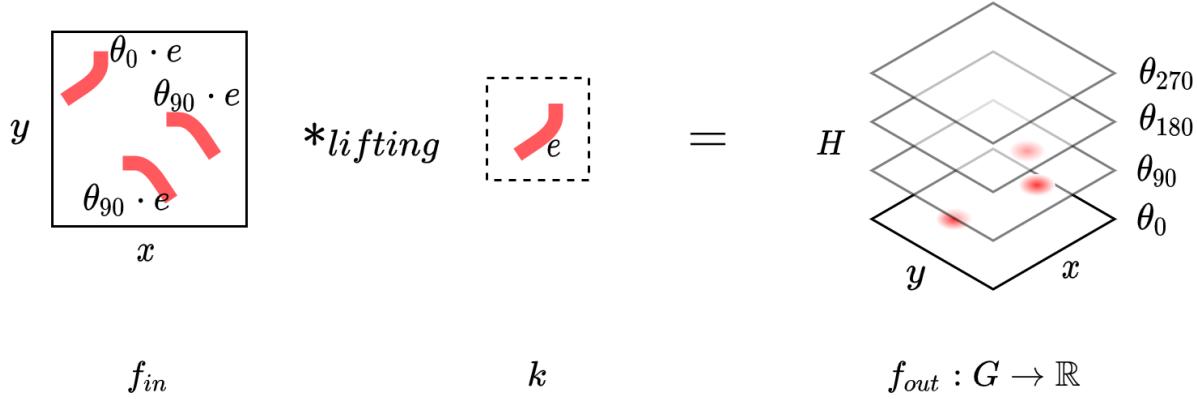


4.30.3 2. Group Equivariant Convolutional Networks

As discussed in the lecture, regular group convolutional neural networks consist of three main elements. The lifting convolution, group convolution and projection operation. We treat these in order.

2.1 Lifting convolution

First is the lifting convolution, which disentangles features at any spatial location in the input f_{in} under transformations of H . You may think of this as registering at all locations, for a given feature e , the occurrences of transformed versions of this feature $\mathcal{L}_h(e)$, for $h \in H$. (Instead of $\mathcal{L}_h(e)$, we sometimes write $h \cdot e$ to denote the action of h on e .) The lifting convolution thus maps from \mathbb{R}^2 to $G = \mathbb{R}^2 \rtimes H$. As a result, our lifted feature map f_{out} has, besides the usual spatial dimensions, one or more additional group dimensions (dependent on the dimensionality of H).



For example, take the group of 90 deg rotations as H , and let's say e is a squiggle of sorts, of which we have three occurrences in our input feature map f_{in} . Two are under a 90 deg rotation; $\theta_{90} \cdot e$, and one is under its canonical orientation; $\theta_0 \cdot e$. A lifting convolution with a kernel k which exactly matches the feature e would land responses at different offsets along the group dimension of the feature map; namely one at the spatial feature map corresponding to the group elements θ_0 and two at the spatial feature map corresponding to θ_{90} . See the above figure for an intuition.

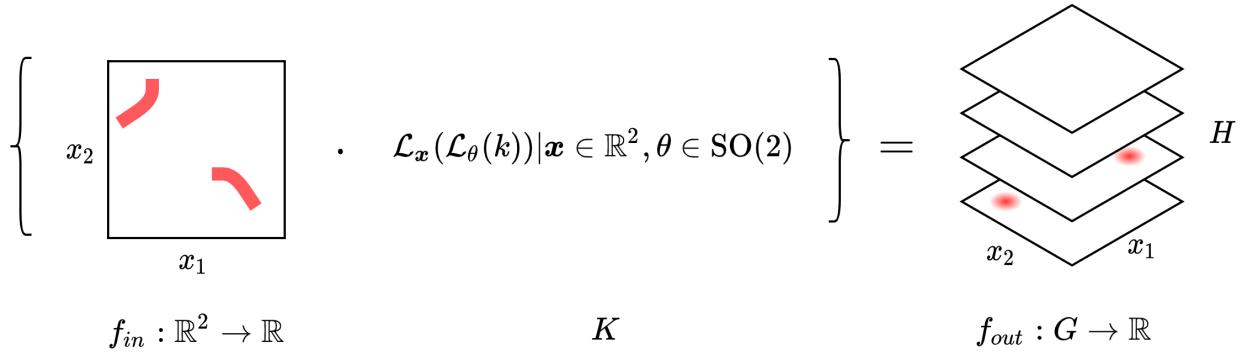
2.1.1 Overview

How do we get our convolution operation to pick up features under different transformations of H ? Intuitively, it is not that different from the convolution operation as we are familiar with in CNNs. There, we enable the extraction of features at any location by sharing the same kernel over all spatial positions.

$$\left\{ \begin{array}{c} x_2 \\ \boxed{\text{red wavy lines}} \\ x_1 \end{array} \right. \cdot \left. \begin{array}{c} \mathcal{L}_x(k) | x \in \mathbb{R}^2 \end{array} \right\} = \left. \begin{array}{c} x_2 \\ \boxed{\text{red dots}} \\ x_1 \end{array} \right.$$

From our current group theoretic perspective, we interpret this as applying all possible translations $x \in \mathbb{R}^2$ to our kernel k and recording the response we get when we take the inner product of the input f_{in} with this transformed

kernel $\mathcal{L}_x(k)$. This operation starts with a feature map defined on \mathbb{R}^2 and also yields a feature map defined over \mathbb{R}^2 . See above.



Now that we additionally want to register features under different group actions \mathcal{L}_h for $h \in H$, we can do so by simply *also* transforming k with all of their group actions and recording the results. For example, in case of the rotation group C_4 , we not only translate, but additionally rotate the kernel k by all possible 90 deg rotations, and record the responses for the resulting transformed kernels!

2.1.2 Implementing the lifting convolution kernel

Let's get to programming. First, we need to define a kernel k which we can transform under arbitrary group actions with \mathcal{L}_h . When working with images, a convolution kernel is generally defined as a set of independently sampled weights W defined over an equidistant discretisation of \mathbb{R}^2 (pixels are evenly spaced).

Recall that we can express the group action of a group H on functions (such as kernels k) defined over \mathbb{R}^2 through the regular representation \mathcal{L}_h . The regular representation transforms the function k through a transformation of the domain of the function k . In other words, the regular representation transforms the grid over which the kernel k is defined, to obtain the values for the transformed function $\mathcal{L}_h(k)$.

As such, to define a kernel k which we can transform with the regular representation of a group H , we need to construct a grid over which the kernel values are defined. We can then transform this grid by the action of each group element $h \in H$ to obtain a set of grids corresponding to transformed kernels for each of the group elements of H . Let's get to work!

Notes: * In implementing the actual lifting and group convolution operations, we will make use of PyTorch's `Conv2D` class. This simplifies our life a lot, since `Conv2D` takes care of translating the kernels k over all input locations. Hence we do not need to implement the action of the translation group ourselves (\mathcal{L}_x), but will still remain translation equivariant! Making our operations compatible with `Conv2D` requires a small amount of trickery, but we will get to that later.

```
[11]: class LiftingKernelBase(torch.nn.Module):

    def __init__(self, group, kernel_size, in_channels, out_channels):
        """
        """
        super().__init__()
        self.group = group

        self.kernel_size = kernel_size
        self.in_channels = in_channels
        self.out_channels = out_channels
```

(continues on next page)

(continued from previous page)

```

# Create spatial kernel grid. These are the coordinates on which our
# kernel weights are defined.
self.register_buffer("grid_R2", torch.stack(torch.meshgrid(
    torch.linspace(-1, 1, self.kernel_size),
    torch.linspace(-1, 1, self.kernel_size),
)).to(self.group.identity.device))

# Transform the grid by the elements in this group.
self.register_buffer("transformed_grid_R2", self.create_transformed_grid_R2())

def create_transformed_grid_R2(self):
    """Transform the created grid by the group action of each group element.
    This yields a grid (over H) of spatial grids (over R2). In other words,
    a list of grids, each index of which is the original spatial grid transformed by
    a corresponding group element in H.

    """
    # Obtain all group elements.
    group_elements = self.group.elements()

    # Transform the grid defined over R2 with the sampled group elements.
    transformed_grid = self.group.left_action_on_R2(
        self.group.inverse(group_elements),
        self.grid_R2
    )
    return transformed_grid

def sample(self, sampled_group_elements):
    """ Sample convolution kernels for a given number of group elements

    arguments should include:
    :param sampled_group_elements: the group elements over which to sample
        the convolution kernels

    should return:
    :return kernels: filter bank extending over all input channels,
        containing kernels transformed for all output group elements.
    """
    raise NotImplementedError()

```

[12]: # Let's visualize the transformed kernel grids!

```

lifting_kernel_base = LiftingKernelBase(
    group=CyclicGroup(order=4),
    kernel_size=7,
    in_channels=1,
    out_channels=1
)

transformed_grid_R2 = lifting_kernel_base.transformed_grid_R2

```

(continues on next page)

(continued from previous page)

```
# The grid has a shape of [num_group_elements, kernel_size, kernel_size, dim_fmap_
˓→domain( $R^2$ )]
transformed_grid_R2.shape

[12]: torch.Size([4, 7, 7, 2])

[13]: plt.rcParams['figure.figsize'] = [12, 3]

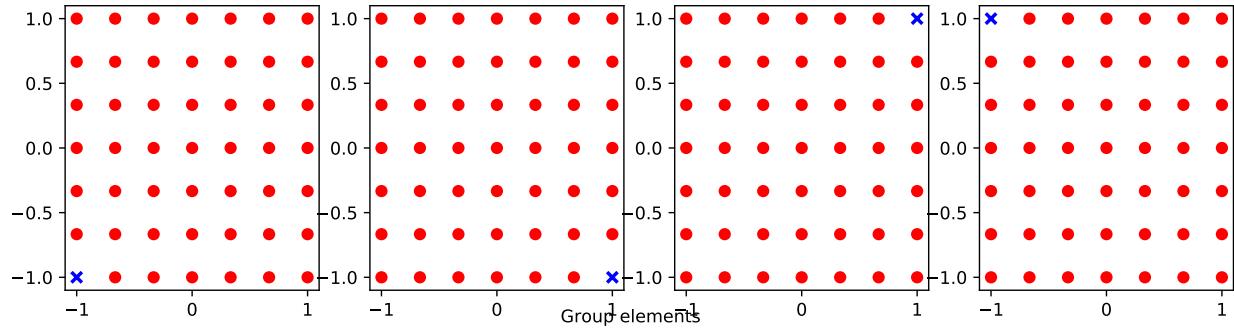
# Create [group_elements] figures
fig, ax = plt.subplots(1, transformed_grid_R2.shape[0])

# Fold both spatial dimensions into a single dimension
transformed_grid_R2 = transformed_grid_R2.reshape(transformed_grid_R2.shape[0],
                                                 transformed_grid_R2.
˓→shape[1]*transformed_grid_R2.shape[2],
                                                 2).numpy()

# Visualize the transformed kernel grids. We mark the same cornerpoint by a blue 'x' in_
˓→all grids as reference point.
for group_elem in range(transformed_grid_R2.shape[0]):
    ax[group_elem].scatter(transformed_grid_R2[group_elem, 0, 0], transformed_grid_
˓→R2[group_elem, 0, 1], marker='x', c='b')
    ax[group_elem].scatter(transformed_grid_R2[group_elem, 1:, 0], transformed_grid_
˓→R2[group_elem, 1:, 1], c='r')

fig.text(0.5, 0.04, 'Group elements', ha='center')

plt.show()
```



The grid transforms under the group action! Pretty cool.

At this point we have a set of grids transformed under the operation of the group H . We now need to decide how we are going to sample kernel values at the grid points in each of these grids. This is where the first major hurdle in applying GCNNs occurs.

Whereas conventional CNNs get away with sharing the same set of weights over all spatial positions (which is due to the fact that we translate the kernel only with steps that match whole pixel-distances), for arbitrary groups H we may require kernel values for grid points that lie off the pixel grid of the kernel under its canonical transformation.

Of course, we could (and in fact will) use interpolation to obtain kernel values for grid locations between pixel locations, but this may be limiting in expressivity and will introduce interpolation artefacts!

Notes:

- When we are implementing equivariance for the group of 90 deg rotations, $H = C_4$, we of course could get away without using interpolation, since all transformed grids lie share the same locations. We could implement the group action of this particular group through a permutation of the weights. But we'd like to be more general than that in this tutorial, so we'll go with interpolation instead! Luckily PyTorch has a function that allows us to sample an input on a grid; we will use PyTorch's `grid_sample` function for interpolation!

```
[14]: class InterpolativeLiftingKernel(LiftingKernelBase):

    def __init__(self, group, kernel_size, in_channels, out_channels):
        super().__init__(group, kernel_size, in_channels, out_channels)

        # create and initialise a set of weights, we will interpolate these
        # to create our transformed spatial kernels.
        self.weight = torch.nn.Parameter(torch.zeros((
            self.out_channels,
            self.in_channels,
            self.kernel_size,
            self.kernel_size
        ), device=self.group.identity.device))

        # Initialize weights using kaiming uniform intialisation
        torch.nn.init.kaiming_uniform_(self.weight.data, a=math.sqrt(5))

    def sample(self):
        """ Sample convolution kernels for a given number of group elements

        should return:
        :return kernels: filter bank extending over all input channels,
                         containing kernels transformed for all output group elements.
        """
        # We fold the output channel dim into the input channel dim; this allows
        # us to use the torch grid_sample function.
        weight = self.weight.view(
            1,
            self.out_channels * self.in_channels,
            self.kernel_size,
            self.kernel_size
        )

        # We want a transformed set of weights for each group element so
        # we repeat the set of spatial weights along the output group axis
        weight = weight.repeat(self.group.elements().numel(), 1, 1, 1)

        # Sample the transformed kernels
        transformed_weight = torch.nn.functional.grid_sample(
            weight,
            self.transformed_grid_R2,
            mode='bilinear',
            padding_mode='zeros',
            align_corners=True
        )

        # Separate input and output channels
        return transformed_weight
```

(continues on next page)

(continued from previous page)

```

transformed_weight = transformed_weight.view(
    self.group.elements().numel(),
    self.out_channels,
    self.in_channels,
    self.kernel_size,
    self.kernel_size
)

# Put the output channel dimension before the output group dimension.
transformed_weight = transformed_weight.transpose(0, 1)

return transformed_weight

```

```
[15]: ik = InterpolativeLiftingKernel(
    group=CyclicGroup(order=4),
    kernel_size=5,
    in_channels=2,
    out_channels=1
)

weights = ik.sample()
```

Let's visualize the weights we sampled from our lifting convolution kernel!

```
[16]: plt.rcParams['figure.figsize'] = [10, 5]

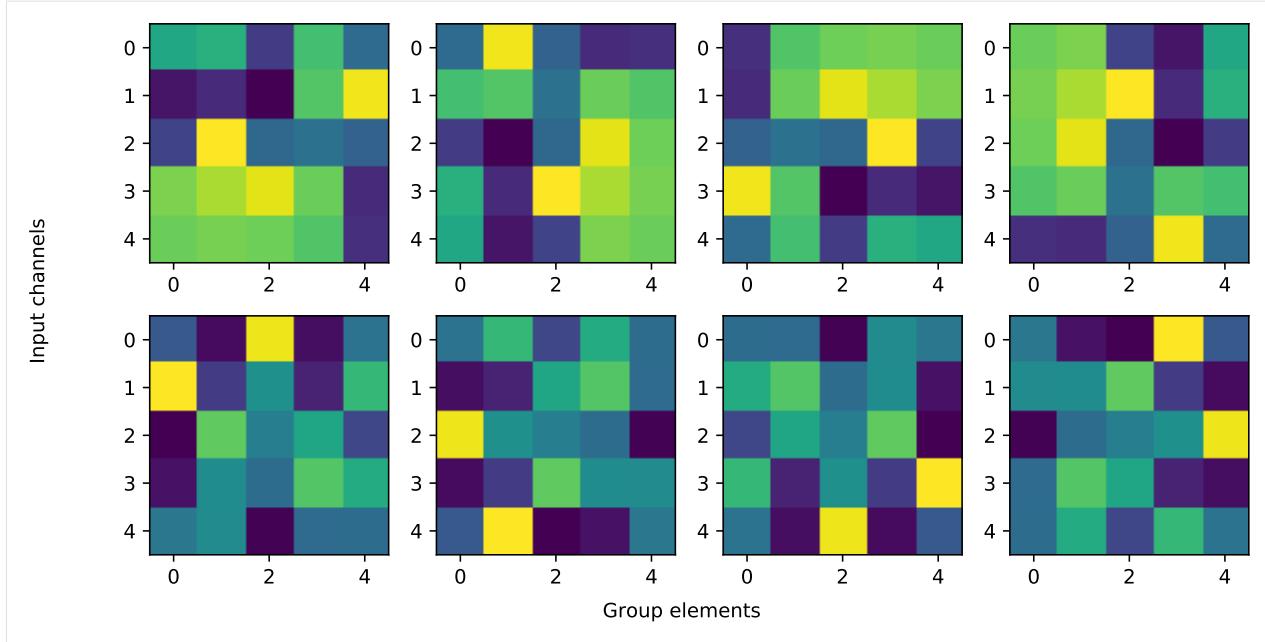
# pick an output channel to visualize
out_channel_idx = 0

# create [in_channels, group_elements] figures
fig, ax = plt.subplots(weights.shape[2], weights.shape[1])

for in_channel in range(weights.shape[2]):
    for group_elem in range(weights.shape[1]):
        ax[in_channel, group_elem].imshow(
            weights[out_channel_idx, group_elem, in_channel, :, :].detach().numpy()
        )

fig.text(0.5, 0.04, 'Group elements', ha='center')
fig.text(0.04, 0.5, 'Input channels', va='center', rotation='vertical')

plt.show()
```



As we can see, spatial kernel rotates under the action of the rotation group elements!

2.1.3 Implementing the lifting convolution

Finally, we can implement the lifting convolution operation! This class should take as input a feature map defined over \mathbb{R}^2 , and spit out a feature map over $\mathbb{R}^2 \times H$, where features under different transformations $h \in H$ are disentangled along the H axis!

Notes:

- To prevent having to implement our own implementation of the convolution operation, and to leverage the highly optimized pytorch Conv2D class, we use some neat tricks in our lifting (and group) convolution classes. Normally, a convolution layer applies a set of n spatial kernels throughout the input, where n is the number of output channels of the convolution operation. Because we now also have `num_group_elem` transformed versions of these kernels, which we want to apply everywhere in the input, we can trick PyTorch by having it treat every transformation of the same spatial kernel as a separate output channel. To do this, we simply reshape our set of `[out_channels, num_group_elem, in_channels, kernel_size, kernel_size]` kernels into a set of `[out_channels x num_group_elem, in_channels, kernel_size, kernel_size]` kernels. See below!
- As mentioned, a great additional benefit of using the PyTorch Conv2D class is that we are not required to obtain translated kernels $\mathcal{L}_x(k)$ ourselves, PyTorch takes care of that!

```
[17]: class LiftingConvolution(torch.nn.Module):

    def __init__(self, group, in_channels, out_channels, kernel_size):
        super().__init__()

        self.kernel = InterpolativeLiftingKernel(
            group=group,
            kernel_size=kernel_size,
            in_channels=in_channels,
            out_channels=out_channels
```

(continues on next page)

(continued from previous page)

```

)
def forward(self, x):
    """ Perform lifting convolution

    @param x: Input sample [batch_dim, in_channels, spatial_dim_1,
        spatial_dim_2]
    @return: Function on a homogeneous space of the group
        [batch_dim, out_channels, num_group_elements, spatial_dim_1,
        spatial_dim_2]
"""

# obtain convolution kernels transformed under the group
conv_kernels = self.kernel.sample()

# apply lifting convolution, note that the reshape folds the group
# dimension of the kernel into the output channel dimension.
# Do you see why we (can) do this?
x = torch.nn.functional.conv2d(
    input=x,
    weight=conv_kernels.reshape(
        self.kernel.out_channels * self.kernel.group.elements().numel(),
        self.kernel.in_channels,
        self.kernel.kernel_size,
        self.kernel.kernel_size
    ),
)

# reshape [batch_dim, in_channels * num_group_elements, spatial_dim_1,
# spatial_dim_2] into [batch_dim, in_channels, num_group_elements,
# spatial_dim_1, spatial_dim_2], separating channel and group
# dimensions.
x = x.view(
    -1,
    self.kernel.out_channels,
    self.kernel.group.elements().numel(),
    x.shape[-1],
    x.shape[-2]
)

return x

```

```
[18]: lifting_conv = LiftingConvolution(
    group=CyclicGroup(order=4),
    kernel_size=5,
    in_channels=3,
    out_channels=8
)
```

2.2 Group convolution

Now that we have a way to obtain feature maps defined over the group G from input functions defined over \mathbb{R}^2 , let's move on to implementing a convolutional layer that fully operates on the group $G : \mathbb{R}^2 \rtimes H$. Note that the input feature map at this stage, f_{in} , has, besides the usual spatial dimensions defined over \mathbb{R}^2 , one or more additional group dimensions defined over H . As such, the group convolution operation, $*_{group}$, maps from a function on the group f_{in} to another function on the group f_{out} .

2.2.1 Overview

Since the input to the group convolution layer now contains additional group dimensions; $f_{in} : \mathbb{R}^2 \rtimes H \rightarrow \mathbb{R}$, we need to convolve it with a kernel k_{group} that is also defined over the entire group; $k_{group} : \mathbb{R}^2 \rtimes H \rightarrow \mathbb{R}$. This is in contrast to the lifting convolution, where $k_{lifting}$ was only defined over the spatial domain \mathbb{R}^2 . You could think of this kernel k_{group} as a stack of spatial kernels, a separate one for each group element $h \in H$. Importantly, since k is now *also* defined on a grid over H , the group H now also has an action *on* this H -axis of k . For example, in our case of $H = C_4$, elements $\theta \in C_4$ now not only have a rotating action on the spatial domain of the kernel, but also a translating action along the group axis. Hence, applying a group element $\theta \in C_4$ leads to a twist-shift of the k along the group axis. See below! (Remember $H = C_4$ is periodic)

$$k : G \rightarrow \mathbb{R} \quad \mathcal{L}_{\mathbf{x}_{(2,2)}} \mathcal{L}_{\theta_{90}} \quad \mathcal{L}_{\mathbf{x}_{(2,2)}} \mathcal{L}_{\theta_{90}}(k) : G \rightarrow \mathbb{R}$$

Apart from this difference, the group convolution operator works in much the same way as the lifting operator. We again transform the kernel k_{group} with the actions of the group H and \mathbb{R}^2 and obtain responses for the inner product of this kernel with the input (although again, we let PyTorch do all the work for the translation group). See below for an intuition.

$$\left\{ \begin{array}{c} \text{Input Feature Map: } f_{in} : G \rightarrow \mathbb{R} \\ \text{Kernel: } K \\ \text{Output Feature Map: } f_{out} : G \rightarrow \mathbb{R} \end{array} \right. \cdot \mathcal{L}_{\mathbf{x}}(\mathcal{L}_{\theta}(k)) | \mathbf{x} \in \mathbb{R}^2, \theta \in SO(2) = \left\{ \begin{array}{c} \text{Output Feature Map: } f_{out} : G \rightarrow \mathbb{R} \end{array} \right.$$

2.2.2 Implementing the group convolution kernel

Again, let's define a kernel k which we can transform with the group action. Now, our kernel grid is not only defined over \mathbb{R}^2 , but additionally over H .

Notes:

- Since the grid over H is made up of elements $h' \in H$, transforming the grid over H with (another) group element $h \in H$ comes out to applying the group product of h with each grid element h' .
- Because we are working with semidirect product groups $\mathbb{R}^2 \rtimes H$, we can transform the \mathbb{R}^2 and H dimensions of our grids separately before combining them into a shared grid over $\mathbb{R}^2 \rtimes H$!

```
[19]: class GroupKernelBase(torch.nn.Module):

    def __init__(self, group, kernel_size, in_channels, out_channels):
        """
        """
        super().__init__()
        self.group = group

        self.kernel_size = kernel_size
        self.in_channels = in_channels
        self.out_channels = out_channels

        # Create a spatial kernel grid
        self.register_buffer("grid_R2", torch.stack(torch.meshgrid(
            torch.linspace(-1, 1, self.kernel_size),
            torch.linspace(-1, 1, self.kernel_size),
        )).to(self.group.identity.device))

        # The kernel grid now also extends over the group H, as our input
        # feature maps contain an additional group dimension
        self.register_buffer("grid_H", self.group.elements())

        self.register_buffer("transformed_grid_R2xH", self.create_transformed_grid_
        ↵R2xH())

    def create_transformed_grid_R2xH(self):
        """
        """
        # Sample the group
        group_elements = self.group.elements()

        # Transform the grid defined over R2 with the sampled group elements
        transformed_grid_R2 = self.group.left_action_on_R2(
            self.group.inverse(group_elements),
            self.grid_R2
        )

        # Transform the grid defined over H with the sampled group elements
        transformed_grid_H = self.group.left_action_on_H(

```

(continues on next page)

(continued from previous page)

```

    self.group.inverse(group_elements), self.grid_H
)

# Rescale values to between -1 and 1, we do this to please the torch grid_sample
# function.
transformed_grid_H = self.group.normalize_group_elements(transformed_grid_H)

# Create a combined grid as the product of the grids over R2 and H
# repeat R2 along the group dimension, and repeat H along the spatial dimension
# to create a [output_group_elem, num_group_elements, kernel_size, kernel_size, ↵
↪3] grid
transformed_grid = torch.cat(
(
    transformed_grid_R2.view(
        group_elements.numel(),
        1,
        self.kernel_size,
        self.kernel_size,
        2,
    ).repeat(1, group_elements.numel(), 1, 1, 1),
    transformed_grid_H.view(
        group_elements.numel(),
        group_elements.numel(),
        1,
        1,
        1,
    ).repeat(1, 1, self.kernel_size, self.kernel_size, 1, )
),
dim=-1
)
return transformed_grid

def sample(self, sampled_group_elements):
    """ Sample convolution kernels for a given number of group elements

    arguments should include:
    :param sampled_group_elements: the group elements over which to sample
        the convolution kernels

    should return:
    :return kernels: filter bank extending over all input channels,
        containing kernels transformed for all output group elements.
    """
    raise NotImplementedError()

```

Let's get some intuition for what is happening with our grid when we apply the group action of H to it. First we will inspect the action on \mathbb{R}^2 .

```
[20]: group_kernel_base = GroupKernelBase(
    group=CyclicGroup(order=4),
    kernel_size=7,
```

(continues on next page)

(continued from previous page)

```

        in_channels=1,
        out_channels=1
    )

# Sample the group
group_elements = group_kernel_base.group.elements()

# Transform the grid defined over R2 with the sampled group elements
transformed_grid_R2 = group_kernel_base.group.left_action_on_R2(
    group_kernel_base.group.inverse(group_elements),
    group_kernel_base.grid_R2
)

```

[21]: plt.rcParams['figure.figsize'] = [12, 3]

```

# Create [group_elements] figures
fig, ax = plt.subplots(1, transformed_grid_R2.shape[0])

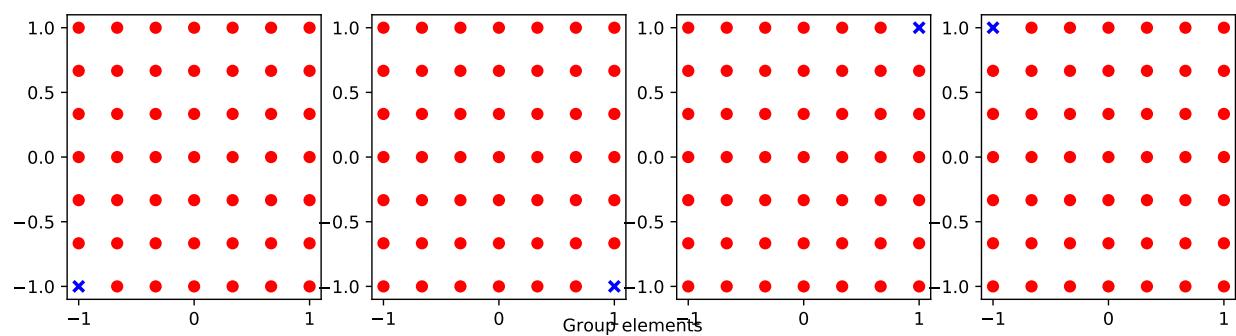
# Fold both spatial dimensions into a single dimension
transformed_grid_R2 = transformed_grid_R2.reshape(transformed_grid_R2.shape[0],
                                                 transformed_grid_R2.
                                                 shape[1]*transformed_grid_R2.shape[2],
                                                 2).numpy()

# Visualize the transformed kernel grids. We mark the same cornerpoint by a blue 'x' in
# all grids as reference point.
for group_elem in range(transformed_grid_R2.shape[0]):
    ax[group_elem].scatter(transformed_grid_R2[group_elem, 0, 0], transformed_grid_
R2[group_elem, 0, 1], marker='x', c='b')
    ax[group_elem].scatter(transformed_grid_R2[group_elem, 1:, 0], transformed_grid_
R2[group_elem, 1:, 1], c='r')

fig.text(0.5, 0.04, 'Group elements', ha='center')

plt.show()

```



As we can see, this part of the grid, and what happens to it, is identical to the grid we saw in the lifting convolution. Note however, that this is only the spatial dimensions of the grid over which a group convolution kernel is defined. Let's move on to the grid over H .

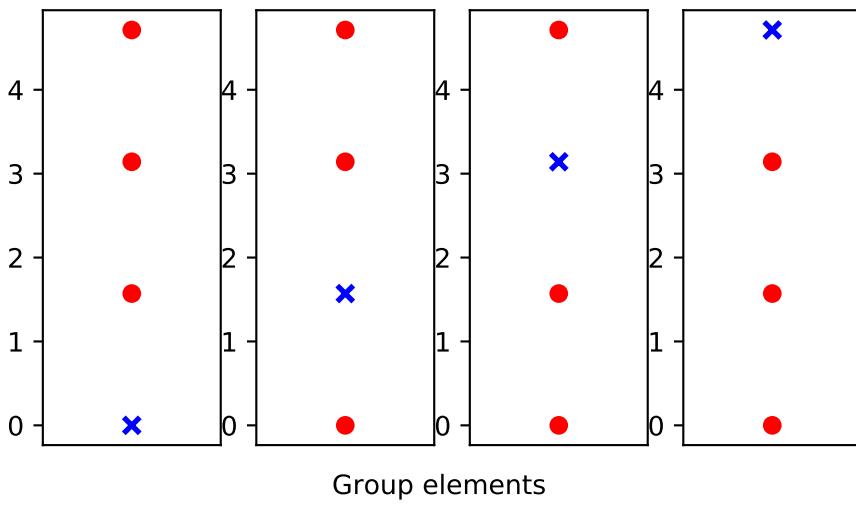
```
[22]: plt.rcParams['figure.figsize'] = [5.5, 3]
# Transform the grid defined over H with the sampled group elements
transformed_grid_H = group_kernel_base.group.left_action_on_H(
    group_kernel_base.group.inverse(group_elements), group_kernel_base.grid_H
)

# Create [group_elements] figures
fig, ax = plt.subplots(1, transformed_grid_H.shape[0])

# Visualize the transformed kernel grids. We mark the same cornerpoint by a blue 'x' in all grids as reference point.
for group_elem in range(transformed_grid_H.shape[0]):
    ax[group_elem].scatter(torch.zeros_like(transformed_grid_H[group_elem, 0]), transformed_grid_H[group_elem, 0], marker='x', c='b')
    ax[group_elem].scatter(torch.zeros_like(transformed_grid_H[group_elem, 1]), transformed_grid_H[group_elem, 1], c='r')
    ax[group_elem].set_xticks([])

fig.text(0.5, 0.04, 'Group elements', ha='center')

# Remove the xticks, our group is 1D!
plt.show()
```



In our current setting, H is one-dimensional, and as we can see, transforming the grid over H with all group elements of H leads to a translation over the group. Next, let's see what happens when we combine these grids.

```
[23]: plt.rcParams['figure.figsize'] = [12, 3]
transformed_grid_R2xH = group_kernel_base.transformed_grid_R2xH

# Create [group_elements] figures
fig, ax = plt.subplots(1, transformed_grid_H.shape[0], subplot_kw=dict(projection='3d'))

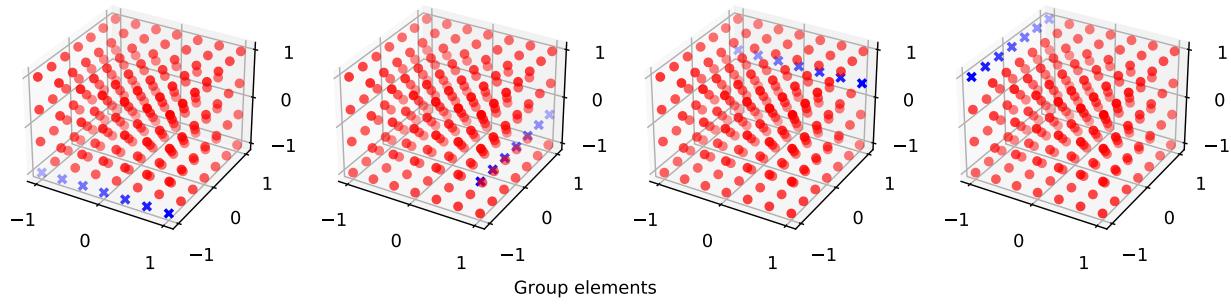
# Flatten spatial and group grid dimensions
transformed_grid_R2xH = transformed_grid_R2xH.reshape(transformed_grid_R2xH.shape[0],
                                                       transformed_grid_R2xH.shape[1] *
                                                       transformed_grid_R2xH.shape[2] * transformed_grid_R2xH.shape[3],
```

(continues on next page)

(continued from previous page)

transformed_grid_R2xH.shape[4])

```
# Visualize the transformed kernel grids. We mark the same row by a blue 'x' in all grids ↵
→ as reference point.
for group_elem in range(transformed_grid_R2xH.shape[0]):
    ax[group_elem].scatter(transformed_grid_R2xH[group_elem, 0:7, 0],
                           transformed_grid_R2xH[group_elem, 0:7, 1],
                           transformed_grid_R2xH[group_elem, 0:7, 2],
                           marker='x',
                           c='b')
    ax[group_elem].scatter(transformed_grid_R2xH[group_elem, 7:, 0],
                           transformed_grid_R2xH[group_elem, 7:, 1],
                           transformed_grid_R2xH[group_elem, 7:, 2],
                           c='r')
fig.text(0.5, 0.04, 'Group elements', ha='center')
plt.show()
```



As we can see, under the application of group elements $h' \in H$ the grid defined over $\mathbb{R}^2 \times H$ rotates over the spatial dimensions, and shifts along the group dimension!

Let's now implement the group kernel using interpolation as well.

Notes: * Luckily, `grid_sample` also supports 3D inputs, so we can continue using it. * For multidimensional groups H the following implementation won't work, as this would require kernels defined on grids with dimensionality > 3, which `grid_sample` does not support. To resolve this, one could implement the sampling of the weights along the H dimension using a translation of the weight matrix along the H dimensions, and only interpolate over the spatial dimensions. This is possible because we don't end up between grid points along the group dimension H (remember the closure constraint of the group product?). * The C_4 group exhibits periodicity along the group axis, so our kernels should too. Although we correctly implemented the group product to reflect this, `grid_sample` doesn't know about the periodicity of the weights in its interpolation. This shouldn't be a problem since, again because of the closure constraint, we should always end up exactly on grid points along the group axis, meaning no interpolation is necessary in that direction. In practice, because of the way `grid_sample` is implemented, we may encounter some small interpolation artefacts because of this.

```
[24]: class InterpolativeGroupKernel(GroupKernelBase):
    def __init__(self, group, kernel_size, in_channels, out_channels):
        super().__init__(group, kernel_size, in_channels, out_channels)

        # create and initialise a set of weights, we will interpolate these
```

(continues on next page)

(continued from previous page)

```

# to create our transformed spatial kernels. Note that our weight
# now also extends over the group H
self.weight = torch.nn.Parameter(torch.zeros(
    self.out_channels,
    self.in_channels,
    self.group.elements().numel(), # this is different from the lifting
    ↪convolution
    self.kernel_size,
    self.kernel_size
), device=self.group.identity.device))

# initialize weights using kaiming uniform intialisation
torch.nn.init.kaiming_uniform_(self.weight.data, a=math.sqrt(5))

def sample(self):
    """ Sample convolution kernels for a given number of group elements

    should return:
    :return kernels: filter bank extending over all input channels,
        containing kernels transformed for all output group elements.
    """
    # fold the output channel dim into the input channel dim; this allows
    # us to use the torch grid_sample function
    weight = self.weight.view(
        1,
        self.out_channels * self.in_channels,
        self.group.elements().numel(),
        self.kernel_size,
        self.kernel_size
    )

    # we want a transformed set of weights for each group element so
    weight = weight.repeat(self.group.elements().numel(), 1, 1, 1, 1)

    # sample the transformed kernels,
    transformed_weight = torch.nn.functional.grid_sample(
        weight,
        self.transformed_grid_R2xH,
        mode='bilinear',
        padding_mode='zeros',
        align_corners=True
    )

    # Separate input and output channels. Note we now have a notion of
    # input and output group dimensions in our weight matrix!
    transformed_weight = transformed_weight.view(
        self.group.elements().numel(), # Output group elements (like in the lifting
        ↪convolutoin
        self.out_channels,
        self.in_channels,
        self.group.elements().numel(), # Input group elements (due to the additional
        ↪dimension of our feature map)

```

(continues on next page)

(continued from previous page)

```

        self.kernel_size,
        self.kernel_size
    )

    # Put the output channel dimension before the output group dimension.
    transformed_weight = transformed_weight.transpose(0, 1)

    return transformed_weight

```

[25]: ik = InterpolativeGroupKernel(
 group=CyclicGroup(order=4),
 kernel_size=5,
 in_channels=2,
 out_channels=8
)

[26]: weights = ik.sample()
 weights.shape

[26]: torch.Size([8, 4, 2, 4, 5, 5])

Let's visualize the sampled group convolution kernels! We visualize our 3D kernels in 2D by folding the input group dimension into the first spatial dimension. In doing so, we create a 2D flattened version of the 3D group convolution kernel, where spatial kernels corresponding to the different group elements lie along the spatial dimension. Each channel goes from [num_group_elem, kernel_size, kernel_size] to [num_group_elem x kernel_size, kernel_size].

To clearly see what happens to the group convolution kernel under transformation of the group H , we outline the spatial kernel corresponding to the first input group element in red. For subsequent transformations we can see this spatial kernel. See below!

[27]: plt.rcParams['figure.figsize'] = [10, 10]

For ease of viewing, we fold the input group dimension into the spatial x dimension
weights_t = weights.view(
 weights.shape[0],
 weights.shape[1],
 weights.shape[2],
 weights.shape[3] * weights.shape[4],
 weights.shape[5]
)

pick an output channel to visualize
out_channel_idx = 0

create [in_channels, group_elements] figures
fig, ax = plt.subplots(weights.shape[2], weights.shape[1])

for in_channel in range(weights.shape[2]):
 for group_elem in range(weights.shape[1]):
 ax[in_channel, group_elem].imshow(
 weights_t[out_channel_idx, group_elem, in_channel, :, :].detach()
)

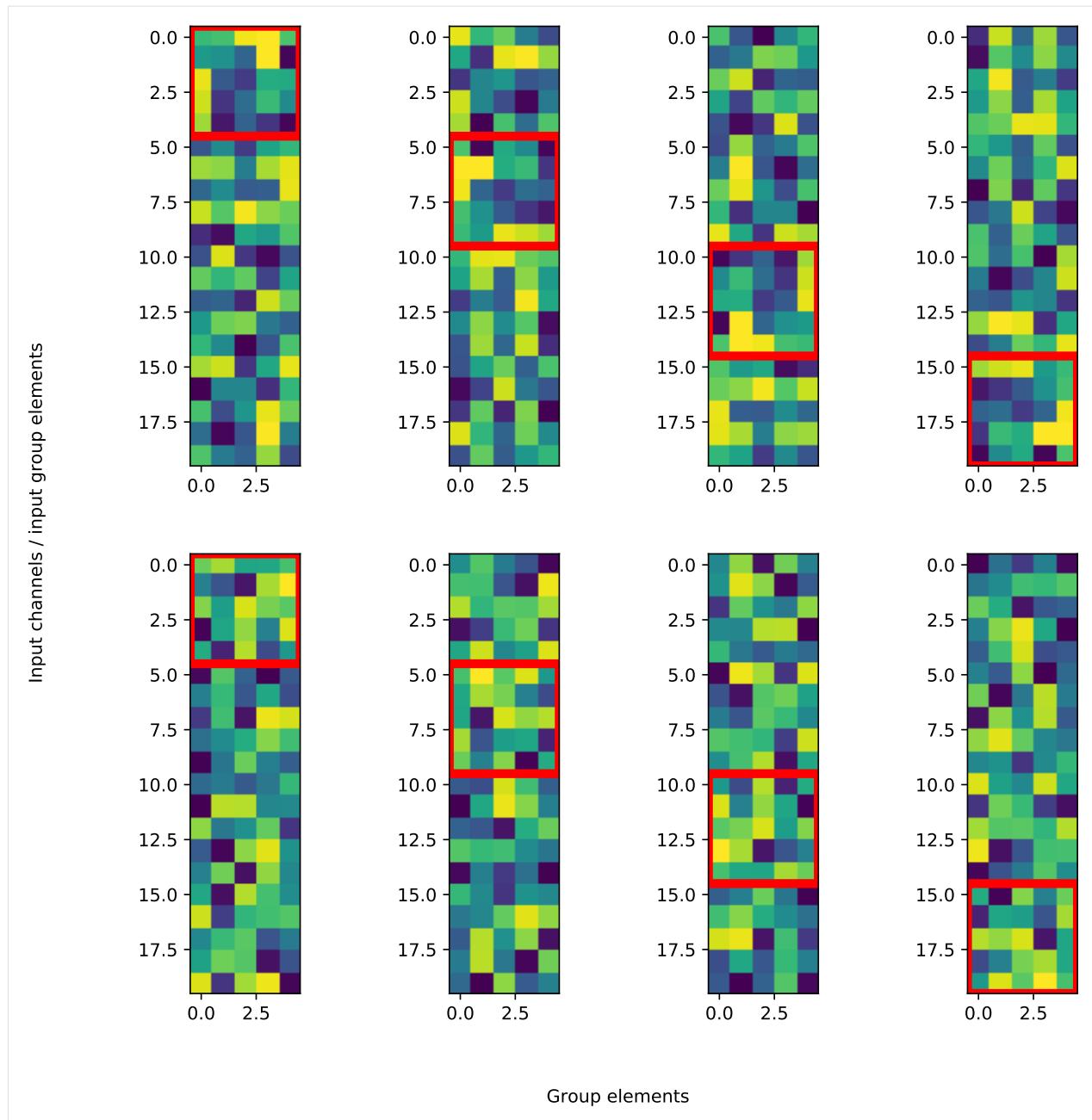
(continues on next page)

(continued from previous page)

```
# Outline the spatial kernel corresponding to the first group element under
# canonical transformation
rect = matplotlib.patches.Rectangle(
    (-0.5, group_elem * weights_t.shape[-1] - 0.5), weights_t.shape[-1], weights_
    t.shape[-1], linewidth=5, edgecolor='r', facecolor='none')
ax[in_channel, group_elem].add_patch(rect)

fig.text(0.5, 0.04, 'Group elements', ha='center')
fig.text(0.04, 0.5, 'Input channels / input group elements', va='center', rotation=
    'vertical')

plt.show()
```



We see the same twist shift motion as we saw with the kernel grids!

2.2.3 Implementing the group convolution

The next step is implementing the group convolution operation.

Notes: * We would still like to use PyTorch's Conv2D implementation, but we're now faced with an additional problem; the group dimension in the input feature map. Luckily we can resolve this problem in much the same way; normally a 2D convolution layer integrates over a local neighbourhood of all input channels. We would now additionally like to integrate over the entire group. Thus, we can simply treat the group dimensions in the input feature map as additional channel dimensions! We achieve this by folding our input group dimension into the input channel dimension; f_{in} is reshaped from [batch, in_channels, num_group_elem, spatial_1, spatial_2] into [batch, in_channels x num_group_elem, spatial_1, spatial_2]. * To match this, and to apply the same trick as we did in the lifting convolution to get results for each separate group element in the *output*, we also reshape our kernel from [out_channels, num_group_elem, in_channels, num_group_elem, kernel_size, kernel_size] to [out_channels x num_group_elem, in_channels x num_group_elem, kernel_size, kernel_size]. See below!

```
[28]: class GroupConvolution(torch.nn.Module):

    def __init__(self, group, in_channels, out_channels, kernel_size):
        super().__init__()

        self.kernel = InterpolativeGroupKernel(
            group=group,
            kernel_size=kernel_size,
            in_channels=in_channels,
            out_channels=out_channels
        )

    def forward(self, x):
        """ Perform lifting convolution

        @param x: Input sample [batch_dim, in_channels, group_dim, spatial_dim_1,
                           spatial_dim_2]
        @return: Function on a homogeneous space of the group
                 [batch_dim, out_channels, num_group_elements, spatial_dim_1,
                  spatial_dim_2]
        """

        # We now fold the group dimensions of our input into the input channel
        # dimension
        x = x.reshape(
            -1,
            x.shape[1] * x.shape[2],
            x.shape[3],
            x.shape[4]
        )

        # We obtain convolution kernels transformed under the group
        conv_kernels = self.kernel.sample()

        # Apply group convolution, note that the reshape folds the group
        # dimension of the kernel into the output channel dimension.
        # Do you see why we (can) do this?

```

(continues on next page)

(continued from previous page)

```

x = torch.nn.functional.conv2d(
    input=x,
    weight=conv_kernels.reshape(
        self.kernel.out_channels * self.kernel.group.elements().numel(),
        self.kernel.in_channels * self.kernel.group.elements().numel(),
        self.kernel.kernel_size,
        self.kernel.kernel_size
    ),
)

# Reshape [batch_dim, in_channels * num_group_elements, spatial_dim_1,
# spatial_dim_2] into [batch_dim, in_channels, num_group_elements,
# spatial_dim_1, spatial_dim_2], separating channel and group
# dimensions.
x = x.view(
    -1,
    self.kernel.out_channels,
    self.kernel.group.elements().numel(),
    x.shape[-1],
    x.shape[-2],
)

```

return x

2.3 Projection to obtain invariance and tying everything together

Up until now, our feature maps equivariant with the group action of $\mathbb{R}^2 \rtimes H$; our feature maps are defined over $\mathbb{R}^2 \rtimes H$. Usually in a CNN, a series of convolutional layers build a representation, which is followed by a (number of) linear layer(s). To create a GCNN using our lifting and group convolution operations that is fully invariant to the action of the group, we must apply a projection operation invariant to the action of the group to our feature map, to reduce its dimensionality from [batch, channels, num_group_elem, spatial_1, spatial_2] to [batch, channels] or even . The representation that we obtain then is fully invariant to the group. This representation is pushed through a final linear layer to yield a classification.

Below, we build a small GCNN from our implemented PyTorch modules.

Notes: * We use a mean-pooling operation to pool over group and spatial dimensions, but we could also use max or min pooling, or any other operation invariant to the group.

```
[29]: class GroupEquivariantCNN(torch.nn.Module):

    def __init__(self, group, in_channels, out_channels, kernel_size, num_hidden, hidden_
    ↪channels):
        super().__init__()

        # Create the lifting convolution.
        self.lifting_conv = LiftingConvolution(
            group=group,
            in_channels=in_channels,
            out_channels=hidden_channels,
            kernel_size=kernel_size

```

(continues on next page)

(continued from previous page)

```

)
# Create a set of group convolutions.
self.gconv = torch.nn.ModuleList()
for i in range(num_hidden):
    self.gconv.append(
        GroupConvolution(
            group=group,
            in_channels=hidden_channels,
            out_channels=hidden_channels,
            kernel_size=kernel_size
        )
)

# Create the projection layer.
self.projection_layer = partial(torch.mean, dim=(-3, -2, -1))

# And a final linear layer for classification.
self.final_linear = torch.nn.Linear(hidden_channels, out_channels)

def forward(self, x):
    # Lift and disentangle features in the input.
    x = self.lifting_conv(x)
    x = torch.nn.functional.layer_norm(x, x.shape[-4:])
    x = torch.nn.functional.relu(x)

    # Apply group convolutions.
    for gconv in self.gconv:
        x = gconv(x)
        x = torch.nn.functional.layer_norm(x, x.shape[-4:])
        x = torch.nn.functional.relu(x)

    # to ensure equivariance, apply max pooling over group and spatial dims
    x = self.projection_layer(x).squeeze()

    x = self.final_linear(x)
    return x

```

To compare, let's create a more or less identical CNN. The only difference here is that this network consists of regular convolution operations.

```
[30]: class CNN(torch.nn.Module):

    def __init__(self, in_channels, out_channels, kernel_size, num_hidden, hidden_
    ↪channels):
        super().__init__()

        self.first_conv = torch.nn.Conv2d(
            in_channels=in_channels,
            out_channels=hidden_channels,
            kernel_size=kernel_size
```

(continues on next page)

(continued from previous page)

```

)
self.convs = torch.nn.ModuleList()
for i in range(num_hidden):
    self.convs.append(
        torch.nn.Conv2d(
            in_channels=hidden_channels,
            out_channels=hidden_channels,
            kernel_size=kernel_size
        )
)

self.final_linear = torch.nn.Linear(hidden_channels, out_channels)

def forward(self, x):
    x = self.first_conv(x)
    x = torch.nn.functional.layer_norm(x, x.shape[-3:])
    x = torch.nn.functional.relu(x)

    for conv in self.convs:
        x = conv(x)
        x = torch.nn.functional.layer_norm(x, x.shape[-3:])
        x = torch.nn.functional.relu(x)

    # apply average pooling over remaining spatial dimensions
    x = torch.nn.functional.adaptive_avg_pool2d(x, 1).squeeze()

    x = self.final_linear(x)
    return x

```

4.30.4 3. Experimenting with our implementation

Note that for ease of model training and tracking, we additionally make use of pytorch-lightning. Although the specifics of this package aren't the focus of this tutorial, if you'd like a refresher please have a look at this excellent Deep Learning 1 tutorial by Phillip Lippe.

3.1 Generalization to the group action

To show the generalization capabilities of regular group convolutional networks, we will train this model on the MNIST training dataset, but evaluate it on an augmented version of the MNIST test set in which each image is randomly rotated by a continuous rotation between $[0, 2\pi]$.

```
[31]: # We normalize the training data.
train_transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.1307,), (0.3081,))
])

# To demonstrate the generalization capabilities our rotation equivariant layers bring, we apply a random
```

(continues on next page)

(continued from previous page)

```
# rotation between 0 and 360 deg to the test set.
test_transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                                torchvision.transforms.RandomRotation(
                                                    [0, 360],
                                                torchvision.transforms.
                                                InterpolationMode.BILINEAR,
                                                fill=0),
                                                torchvision.transforms.Normalize((0.
                                                1307,), (0.3081,)))
                                                ])

# We demonstrate our models on the MNIST dataset.
train_ds = torchvision.datasets.MNIST(root=DATASET_PATH, train=True, transform=train_
    ↪transform, download=True)
test_ds = torchvision.datasets.MNIST(root=DATASET_PATH, train=False, transform=test_
    ↪transform)
train_loader = torch.utils.data.DataLoader(train_ds, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_ds, batch_size=64, shuffle=False)

# Set the random seed for reproducibility.
pl.seed_everything(12)
```

```
Global seed set to 12
```

```
[31]: 12
```

Let's visualize some of the training and test images. As we can see the test images are randomly rotated.

```
[32]: NUM_IMAGES = 4
images = [train_ds[idx][0] for idx in range(NUM_IMAGES)]
orig_images = [Image.fromarray(train_ds.data[idx].numpy()) for idx in range(NUM_IMAGES)]
orig_images = [test_transform(img) for img in orig_images]

img_grid = torchvision.utils.make_grid(torch.stack(images + orig_images, dim=0), nrow=4, ↪
    ↪normalize=True, pad_value=0.5)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("Images sampled from the MNIST train set, augmented with test transforms.")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

Images sampled from the MNIST train set, augmented with test transforms.



```
[33]: class DataModule(pl.LightningModule):

    def __init__(self, model_name, model_hparams, optimizer_name, optimizer_hparams):
        """
        Inputs:
            model_name - Name of the model/CNN to run. Used for creating the model (see
        ↪function below)
            model_hparams - Hyperparameters for the model, as dictionary.
            optimizer_name - Name of the optimizer to use. Currently supported: Adam, SGD
            optimizer_hparams - Hyperparameters for the optimizer, as dictionary. This
        ↪includes learning rate, weight decay, etc.
        """
        super().__init__()
        # Exports the hyperparameters to a YAML file, and create "self.hparams" namespace
        self.save_hyperparameters()
        # Create model
        self.model = create_model(model_name, model_hparams)
        # Create loss module
        self.loss_module = nn.CrossEntropyLoss()

    def forward(self, imgs):
        return self.model(imgs)

    def configure_optimizers(self):
        # AdamW is Adam with a correct implementation of weight decay (see here for
        ↪details: https://arxiv.org/pdf/1711.05101.pdf)
        optimizer = optim.AdamW(
            self.parameters(), **self.hparams.optimizer_hparams)
        return [optimizer], []
```

(continues on next page)

(continued from previous page)

```

def training_step(self, batch, batch_idx):
    # "batch" is the output of the training data loader.
    imgs, labels = batch
    preds = self.model(imgs)
    loss = self.loss_module(preds, labels)
    acc = (preds.argmax(dim=-1) == labels).float().mean()

    # Logs the accuracy per epoch to tensorboard (weighted average over batches)
    self.log('train_acc', acc, on_step=False, on_epoch=True)
    self.log('train_loss', loss)
    return loss # Return tensor to call ".backward" on

def validation_step(self, batch, batch_idx):
    imgs, labels = batch
    preds = self.model(imgs).argmax(dim=-1)
    acc = (labels == preds).float().mean()
    # By default logs it per epoch (weighted average over batches)
    self.log('val_acc', acc, prog_bar=True)

def test_step(self, batch, batch_idx):
    imgs, labels = batch
    preds = self.model(imgs).argmax(dim=-1)
    acc = (labels == preds).float().mean()
    # By default logs it per epoch (weighted average over batches), and returns it afterwards
    self.log('test_acc', acc, prog_bar=True)

```

We create a dictionary to keep track of our models

```
[34]: model_dict = {
    'CNN': CNN,
    'GCNN': GroupEquivariantCNN
}

def create_model(model_name, model_hparams):
    if model_name in model_dict:
        return model_dict[model_name](**model_hparams)
    else:
        assert False, f"Unknown model name \'{model_name}\'. Available models are:
→{str(model_dict.keys())}"

```

```
[35]: def train_model(model_name, save_name=None, **kwargs):
    """
    Inputs:
        model_name - Name of the model you want to run. Is used to look up the class in
→"model_dict"
        save_name (optional) - If specified, this name will be used for creating the
→checkpoint and logging directory.
    """
    if save_name is None:
        save_name = model_name

```

(continues on next page)

(continued from previous page)

```

# Create a PyTorch Lightning trainer with the generation callback
trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, save_name),
                     # Where to save models
                     accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                     # We run on a GPU (if possible)
                     devices=1,
                     # Running on a single GPU/CPU
                     max_epochs=10,
                     # How many epochs to train for if no patience is set
                     callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
                     monitor="val_acc"), # Save the best checkpoint based on the maximum val_acc recorded.
                     # Saves only weights and not optimizer
                     LearningRateMonitor("epoch")])
trainer.logger._default_hp_metric = None # Optional logging argument that we don't
# need

# Check whether pretrained model exists. If yes, load it and skip training
pretrained_filename = os.path.join(CHECKPOINT_PATH, save_name + ".ckpt")

if os.path.isfile(pretrained_filename):
    print(f"Found pretrained model at {pretrained_filename}, loading...")
    model = DataModule.load_from_checkpoint(pretrained_filename) # Automatically
# loads the model with the saved hyperparameters
else:
    pl.seed_everything(12) # To be reproducible
    model = DataModule(model_name=model_name, **kwargs)
    trainer.fit(model, train_loader, test_loader)
    model = DataModule.load_from_checkpoint(trainer.checkpoint_callback.best_model_
# path) # Load best checkpoint after training

# Test best model on test set
val_result = trainer.test(model.to(device), test_loader, verbose=False)
result = {"val": val_result[0]["test_acc"]}

return model, result

```

Let's train the conventional CNN first!

```
[36]: cnn_model, cnn_results = train_model(model_name="CNN",
                                         model_hparams={"in_channels": 1,
                                                        "out_channels": 10,
                                                        "kernel_size": 5,
                                                        "num_hidden": 4,
                                                        "hidden_channels": 32},
                                         optimizer_name="Adam",
                                         optimizer_hparams={"lr": 1e-2,
                                                            "weight_decay": 1e-4},
                                         save_name='cnn-pretrained')
```

GPU available: True, used: True

TPU available: False, using: 0 TPU cores

Found pretrained model at ../../saved_models/DL2/GDL/cnn-pretrained.ckpt, loading...

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
/home/phillip/anaconda3/envs/dl2020/lib/python3.7/site-packages/pytorch_lightning/
↳ utilities/distributed.py:69: UserWarning: The dataloader, test dataloader 0, does not
↳ have many workers which may be a bottleneck. Consider increasing the value of the `num_
↳ workers` argument` (try 16 which is the number of cpus on this machine) in the
↳ `DataLoader` init to improve performance.
    warnings.warn(*args, **kwargs)

Testing: 0it [00:00, ?it/s]
```

Next, we train the GCNN. Note that we reduce the number of channels by half to account for the increased kernel dimensionality. We do this to keep the number of trainable parameters more or less equal.

```
[37]: gcnn_model, gcnn_results = train_model(model_name="GCNN",
                                             model_hparams={"in_channels": 1,
                                                            "out_channels": 10,
                                                            "kernel_size": 5,
                                                            "num_hidden": 4,
                                                            "hidden_channels":16, # to account
                                             ↳ for the increase in trainable parameters due to the extra dimension in our feature
                                             ↳ maps, remove some hidden channels.
                                                       "group":CyclicGroup(order=4).
                                             ↳ to(device)},
                                             optimizer_name="Adam",
                                             optimizer_hparams={"lr": 1e-2,
                                                               "weight_decay": 1e-4},
                                             save_name='gcnn-pretrained')
```

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]

Found pretrained model at ../../saved_models/DL2/GDL/gcnn-pretrained.ckpt, loading...

Testing: 0it [00:00, ?it/s]

```
[38]: %%html
<!-- Some HTML code to increase font size in the following table -->
<style>
th {font-size: 120%;}
td {font-size: 120%;}
</style>

<IPython.core.display.HTML object>
```

Let us inspect the final results from both models. As we can see the GCNN performs considerably better than the CNN. This because the GCNN implementation is invariant to rotations of 90 deg, which makes it able to recognize the handwritten digits in the test set under such rotations. Of course, since the test images are rotated by continuous rotations between 0 and 360 deg, the GCNN model still doesn't obtain perfect accuracy. How could we further improve the GCNNs generalization?

```
[39]: import tabulate
from IPython.display import display, HTML
all_models = [
    ("CNN", cnn_results, cnn_model),
```

(continues on next page)

(continued from previous page)

```

        ("GCNN", gcnn_results, gcnn_model),
    ]
table = [[model_name,
          f'{100.0*model_results['val']:.2f}%',
          '{:,}'.format(sum([np.prod(p.shape) for p in model.parameters()]))]
         for model_name, model_results, model in all_models]
display(HTML(tabulate.tabulate(table, tablefmt='html', headers=["Model", "Val Accuracy",
                                                               "Num Parameters"])))
<IPython.core.display.HTML object>

```

3.2 Inspecting the created feature maps

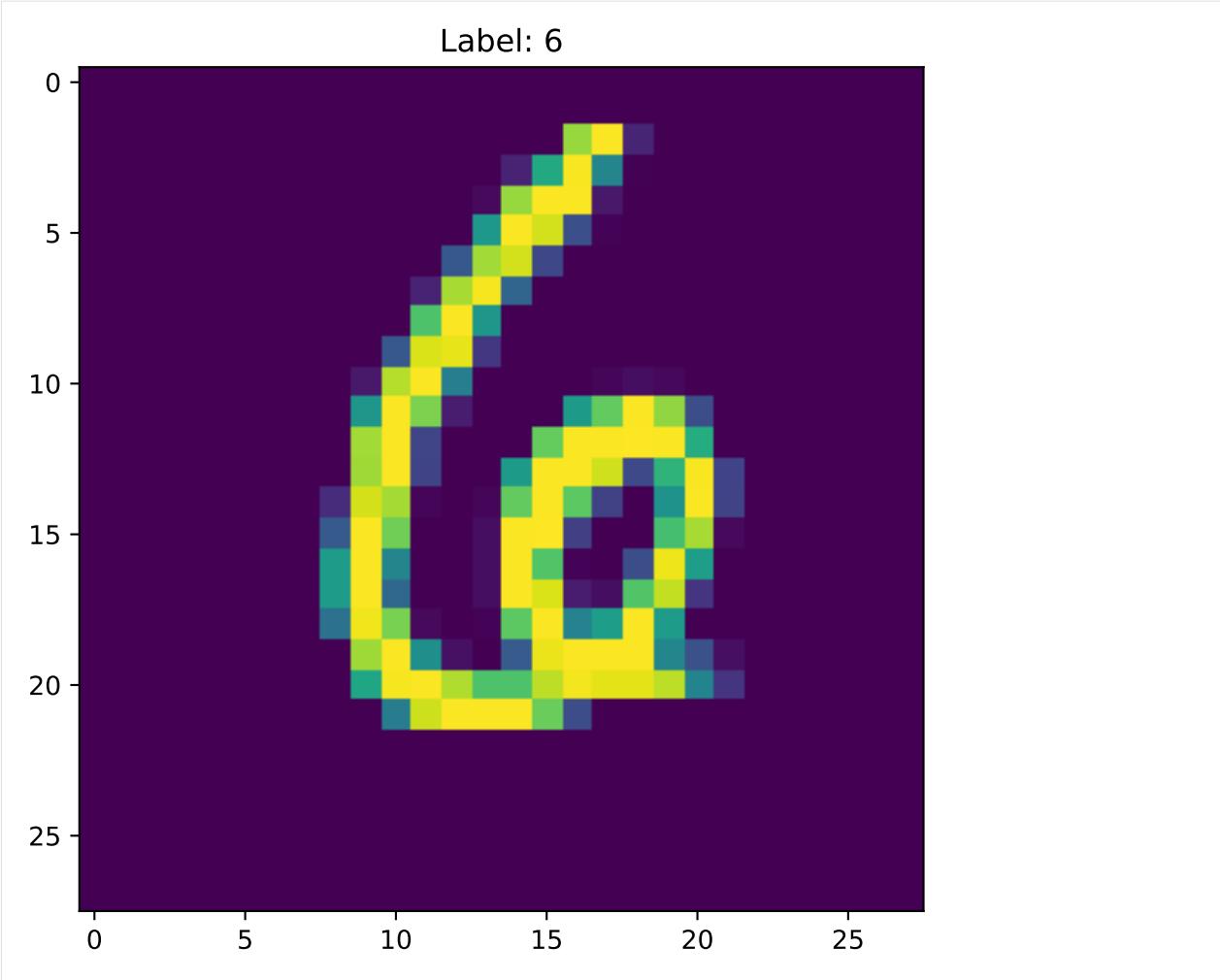
To better understand what happens inside our CNN and GCNN as we rotate an input image, let's visualize a single channel of a feature map in the second layer of the network for different rotations of an input image.

```
[40]: train_ds = torchvision.datasets.MNIST(root=DATASET_PATH, train=False, transform=None)

# Get an image from the test dataset.
digit, label = train_ds[123]

# Turn it into a tensor
digit = transforms.ToTensor()(digit)

plt.figure(figsize=(6, 6))
plt.imshow(digit.squeeze())
plt.title(f'Label: {label}')
plt.show()
```



```
[41]: plt.rcParams['figure.figsize'] = [20, 3]

# Get a set of angles by which to rotate this image.
rots = torch.linspace(0, 360 - 360/8, 8)

# Rotate the input image and push it through the normalization transform
rot_digit = torch.stack(tuple(torchvision.transforms.functional.rotate(digit, a.item(),  
                           torchvision.transforms.functional.InterpolationMode.BILINEAR) for a in rots))
rot_digit = torchvision.transforms.Normalize((0.1307,), (0.3081,))(rot_digit)

# Create a subplot for every rotated input
fig, ax = plt.subplots(1, rots.numel())

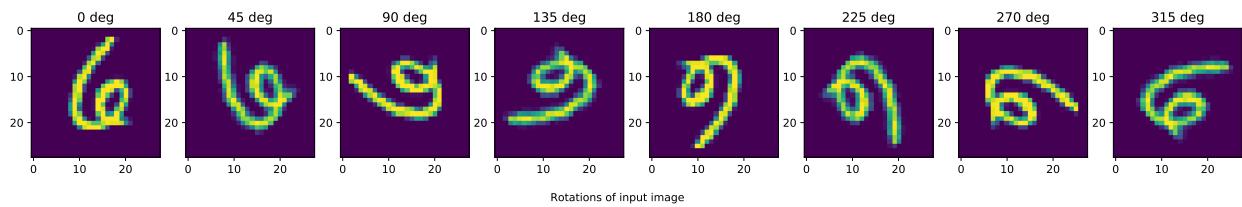
for idx, rotation in enumerate(rots):
    ax[idx].imshow(
        rot_digit[idx, :, :].squeeze()
    )
    ax[idx].set_title(f"int(rotation) deg")

fig.text(0.5, 0.04, 'Rotations of input image', ha='center')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



```
[42]: # Forward it through the first few layers of the CNN
cnn_out = cnn_model.model.first_conv(rot_digit)
cnn_out = torch.nn.functional.relu(torch.nn.functional.layer_norm(cnn_out, cnn_out.
    →shape[-3:]))
for i in range(2):
    cnn_out = cnn_model.model.convs[i](cnn_out)
    cnn_out = torch.nn.functional.relu(torch.nn.functional.layer_norm(cnn_out, cnn_out.
        →shape[-3:]))
```

Let's also see what happens after we apply projection over remaining spatial dimensions.

```
projected_cnn_out = torch.nn.functional.adaptive_avg_pool2d(cnn_out, 1).squeeze()
```

Forward it through the first few layers of the GCNN

```
gcnn_out = gcnn_model.model.lifting_conv(rot_digit)
gcnn_out = torch.nn.functional.relu(torch.nn.functional.layer_norm(gcnn_out, gcnn_out.
    →shape[-4:]))
for i in range(2):
    gcnn_out = gcnn_model.model.gconv[i](gcnn_out)
    gcnn_out = torch.nn.functional.relu(torch.nn.functional.layer_norm(gcnn_out, gcnn_.
        →out.shape[-4:]))
```

And let's see what happens if we apply the projection on this equivariant representation

```
projected_gcnn_out = torch.mean(gcnn_out, dim=(-3, -2, -1))
```

Let's first visualize the activations after the third convolution for a single channel of the regular CNN.

```
[43]: plt.rcParams['figure.figsize'] = [20, 4]

# Pick a channel to visualize
channel_idx = 2

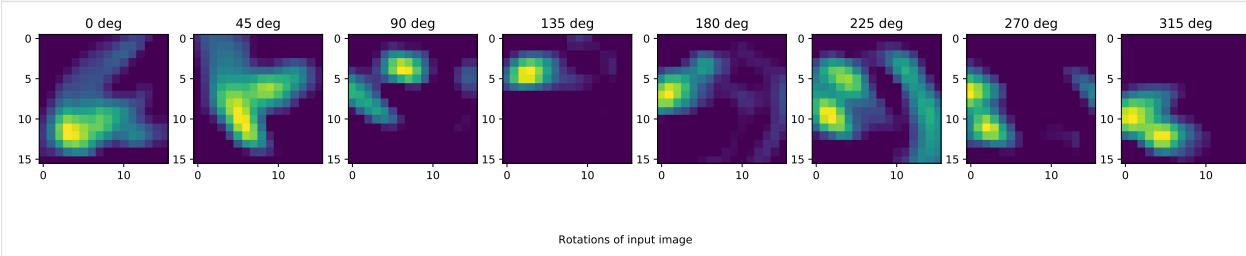
# Create a subfigure for every rotated input
fig, ax = plt.subplots(1, rots.numel())

for idx, rotation in enumerate(rots):
    ax[idx].imshow(
        cnn_out[idx, out_channel_idx, :, :].detach().numpy()
    )
    ax[idx].set_title(f"{int(rotation)} deg")

fig.text(0.5, 0.04, 'Rotations of input image', ha='center')
```

(continues on next page)

(continued from previous page)

`plt.show()`

Representations change dramatically as the input rotates. To the CNN, each of these images are completely different. These representations are not equivariant to rotation.

Let's move on to visualizing a single channel of the GCNN after the second gconv.

```
[44]: plt.rcParams['figure.figsize'] = [20, 9]

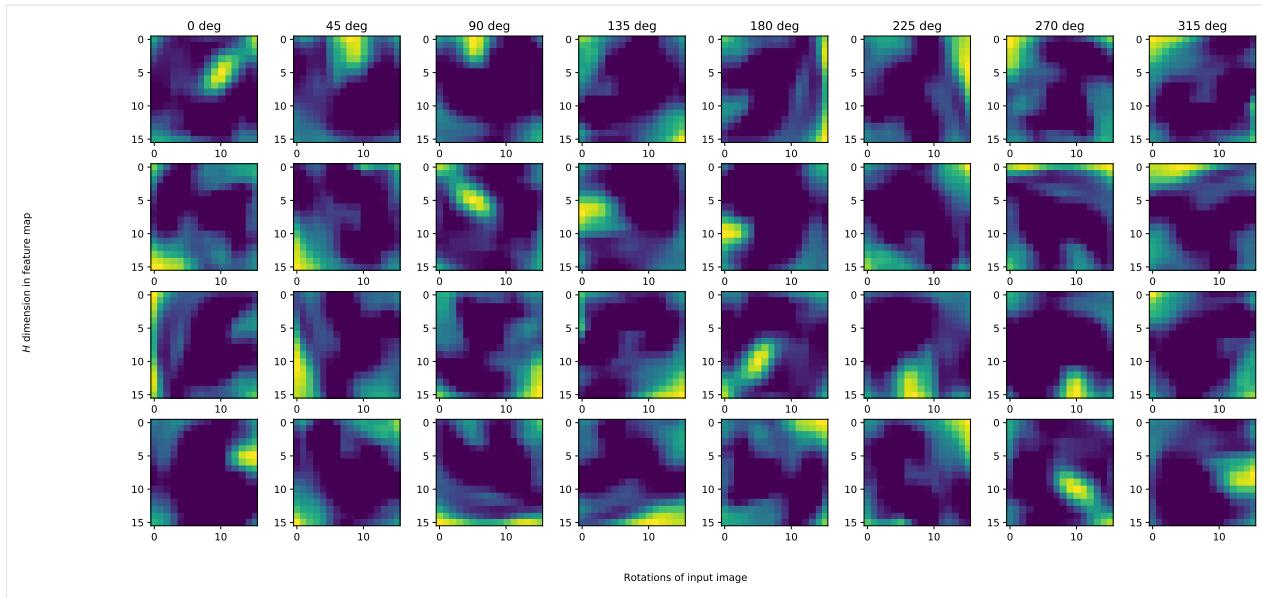
# Pick a channel to visualize
channel_idx = 2

# Create a subfigure for every rotated input and every group element
fig, ax = plt.subplots(gcnn_out.shape[2], rots.numel())

for idx, rotation in enumerate(rots):
    for group_element_idx in range(gcnn_out.shape[2]):
        ax[group_element_idx, idx].imshow(
            gcnn_out[idx, out_channel_idx, group_element_idx, :, :].detach().numpy()
        )
    ax[0, idx].set_title(f"{int(rotation)} deg")

fig.text(0.5, 0.04, 'Rotations of input image', ha='center')
fig.text(0.04, 0.5, '$H$ dimension in feature map', va='center', rotation='vertical')

plt.show()
```



We can see that indeed, save for some interpolation artefacts, the feature maps equivariant with rotations that are multiples of 90 deg! These representations are equivariant to rotations of 90 degrees.

Lastly, we inspect the representations we obtain after the projection step for the CNN and GCNN. We visualize all channels. These are the representations on which the final linear layer(s) perform classification.

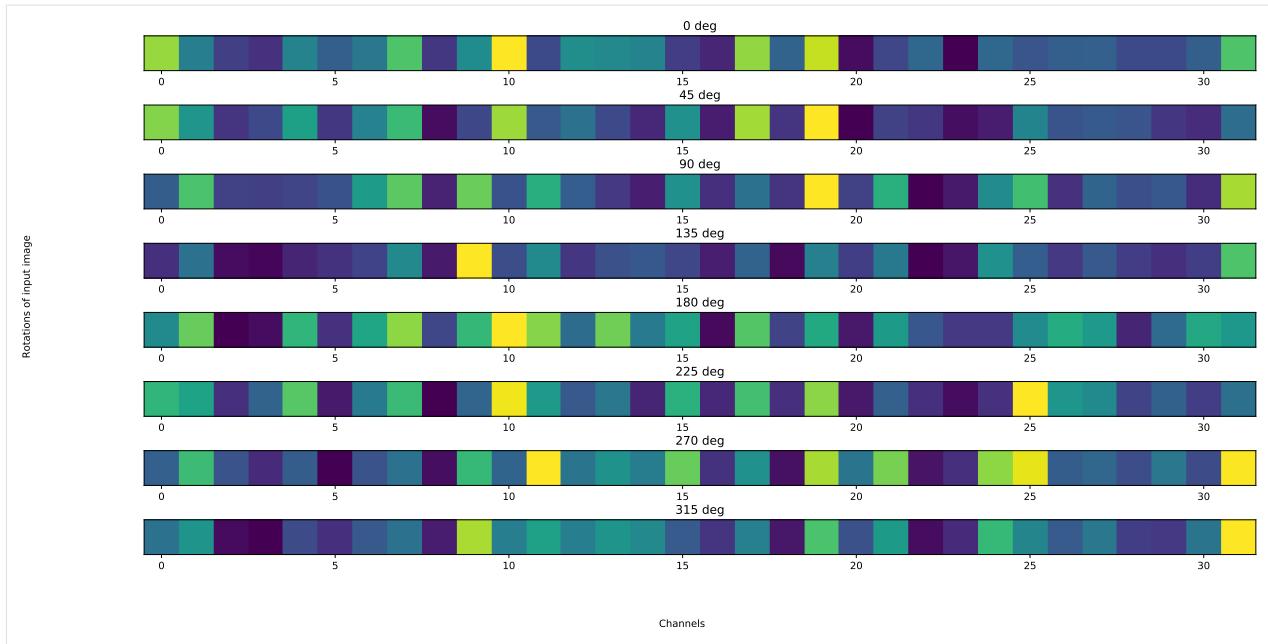
```
[45]: plt.rcParams['figure.figsize'] = [20, 10]

# Create a subfigure for every rotated input
fig, ax = plt.subplots(rots.numel(), 1)

for idx, rotation in enumerate(rots):
    ax[idx].imshow(
        projected_cnn_out[idx, None, :].detach().numpy()
    )
    ax[idx].set_title(f"{int(rotation)} deg")
    ax[idx].set_yticks([])

fig.text(0.5, 0.04, 'Channels', ha='center')
fig.text(0.04, 0.5, 'Rotations of input image', va='center', rotation='vertical')

plt.show()
```



As we can see, as the input image rotates, the representations change. The representations built by the network are not invariant to rotation, and as such, the linear layers may classify them differently.

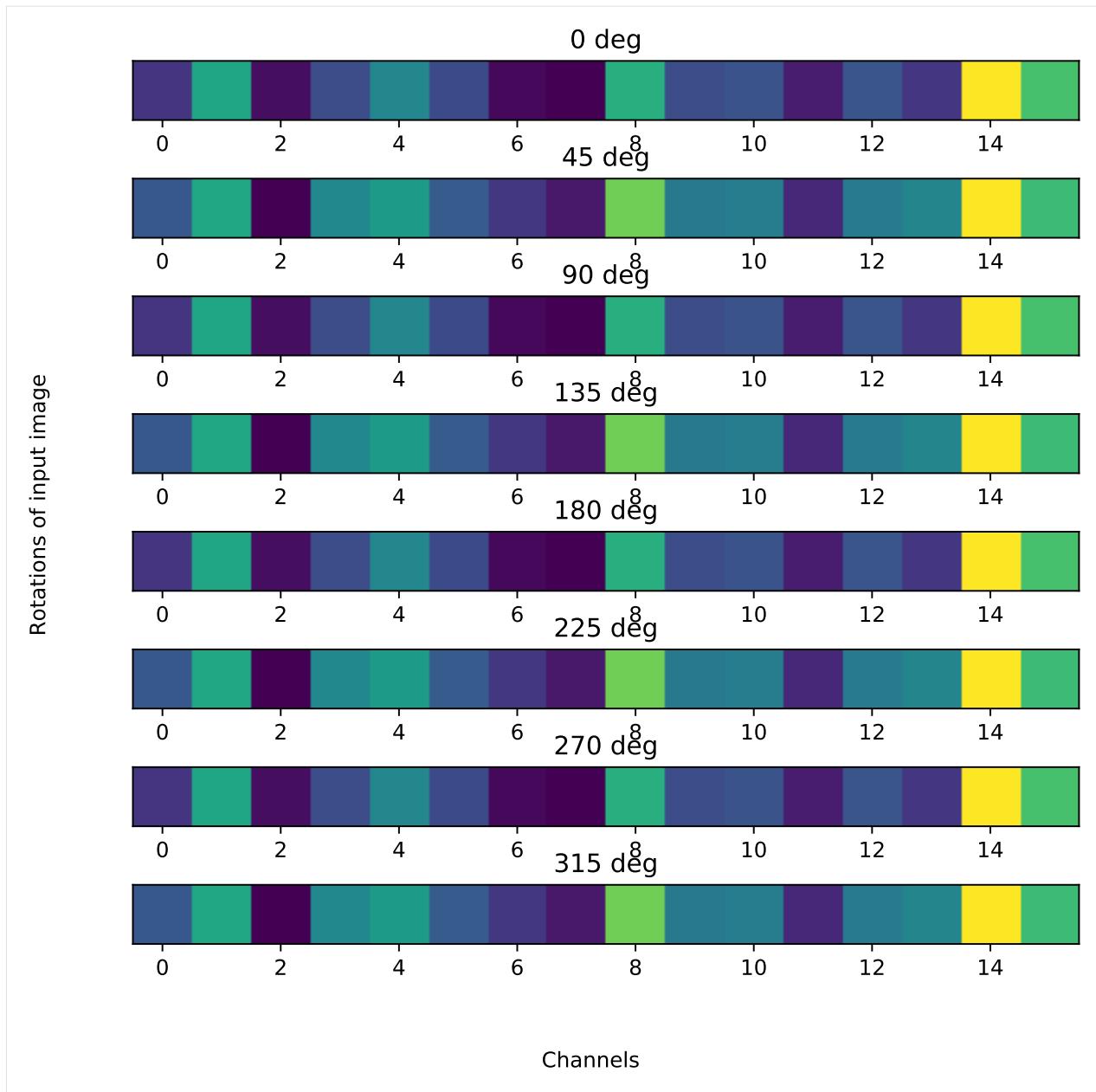
```
[46]: plt.rcParams['figure.figsize'] = [8, 8]

# Create a subfigure for every rotated input
fig, ax = plt.subplots(rots.numel(), 1)

for idx, rotation in enumerate(rots):
    ax[idx].imshow(
        projected_gcnn_out[idx, None, :].detach().numpy()
    )
    ax[idx].set_title(f'{int(rotation)} deg')
    ax[idx].set_yticks([])

fig.text(0.5, 0.04, 'Channels', ha='center')
fig.text(0.04, 0.5, 'Rotations of input image', va='center', rotation='vertical')

plt.show()
```



The representations built for each of the rotated inputs is (more or less) equal! These representations *are* invariant to 90 degree rotations, and the final linear layers see them as identical!

4.30.5 4. Concluding remarks

We've seen how to implement a basic regular group convolutional network equivariant to rotations of 90 degrees. A couple suggestions of things to ponder on... Feel free to use these as starting point for a possible project you'd like to do:

- What other groups could be interesting in computer vision problems? (these often come with their own interesting challenges).
- We used interpolation to obtain our values for transformed kernels. As mentioned, this makes our networks prone to interpolation artefacts. Would our interpolation implementation work well for other groups? Could you maybe think of other ways of defining the kernel over a continuous domain?
- Here we worked with a one-dimensional group. What would it take to implement group convolutions for multi-dimensional groups H ?

4.31 GDL - Steerable CNNs

Filled notebook:

Empty notebook:

Authors: Gabriele Cesa

During the lectures, you have learnt that the symmetries of a machine learning task can be modelled with **groups**. In the previous tutorial, you have also studied the framework of *Group-Convolutional Neural Networks (GCNNs)*, which describes a neural architecture design equivariant to general groups.

The feature maps of a GCNN are functions over the elements of the group. A naive implementation of group-convolution requires computing and storing a response for each group element. For this reason, the GCNN framework is not particularly convenient to implement networks equivariant to groups with infinite elements.

Steerable CNNs are a more general framework which solves this issue. The key idea is that, instead of storing the value of a feature map on each group element, the model stores the *Fourier transform* of this feature map, up to a finite number of frequencies.

In this tutorial, we will first introduce some Representation theory and Fourier theory (*non-commutative harmonic analysis*) and, then, we will explore how this idea is used in practice to implement Steerable CNNs.

4.31.1 Prerequisite Knowledge

Throughout this tutorial, we will assume you are already familiar with some concepts of **group theory**, such as *groups*, *group actions* (in particular *on functions*), *semi-direct product* and *order of a group*, as well as basic **linear algebra**.

We start by importing the necessary packages. You can run the following command to install all the requirements:

```
> pip install torch torchvision numpy matplotlib escnn scipy
```

```
[1]: import torch
import numpy as np
import scipy
import os

np.set_printoptions(precision=3, suppress=True, linewidth=10000, threshold=100000)
```

(continues on next page)

(continued from previous page)

```

import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
# If the fonts in the plots are incorrectly rendered, comment out the next two lines
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
matplotlib.rcParams['lines.linewidth'] = 2.0

import urllib.request
from urllib.error import HTTPError

CHECKPOINT_PATH = ".../.../saved_models/DL2/GDL"

/tmp/ipykernel_31252/3943199899.py:12: DeprecationWarning: `set_matplotlib_formats` is
  deprecated since IPython 7.23, directly use `matplotlib_inline.backend_inline.set_
  matplotlib_formats()`
    set_matplotlib_formats('svg', 'pdf') # For export

```

```

[2]: # Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# Files to download
pretrained_files = [
    "steerable_c4-pretrained.ckpt",
    "steerable_so2-pretrained.ckpt",
    "steerable_c4-accuracies.npy",
    "steerable_so2-accuracies.npy",
]

# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/DL2/GDL/"

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please contact the author with the full output
      including the following error:\n", e)

```

Downloading https://raw.githubusercontent.com/phlippe/saved_models/main/DL2/GDL/
 ↵steerable_c4_accuracies.npy...

Something went wrong. Please contact the author with the full output including the
 ↵following error:

HTTP Error 404: Not Found

Downloading https://raw.githubusercontent.com/phlippe/saved_models/main/DL2/GDL/
 ↵steerable_so2_accuracies.npy...

Something went wrong. Please contact the author with the full output including the
 ↵following error:

(continues on next page)

(continued from previous page)

HTTP Error 404: Not Found

4.31.2 1. Representation Theory and Harmonic Analysis of Compact Groups

We will make use of the `escnn` library throughout this tutorial. You can also find its documentation [here](#).

```
[3]: try:
    from escnn.group import *
except ModuleNotFoundError: # Google Colab does not have escnn installed by default.
    # Hence, we do it here if necessary
    !pip install --quiet escnn
    from escnn.group import *
```

First, let's create a group. We will use the *Cyclic Group* $G = C_8$ as an example. This group contains the 8 planar rotations by multiples of $\frac{2\pi}{8}$. In `escnn`, groups are instances of the abstract class `escnn.group.Group`, which provides some useful functionalities. We instantiate groups via a *factory method*. To build the cyclic group of order 8, we use this factory method:

```
[4]: G = cyclic_group(N=8)

# We can verify that the order of this group is 8:
G.order()
```

```
[4]: 8
```

A group is a collection of group elements together with a binary operation to combine them. This is implemented in the class `escnn.group.GroupElement`. We can access the *identity* element $e \in G$ as

```
[5]: G.identity
```

```
[5]: 0[2pi/8]
```

or sample a random element as

```
[6]: G.sample()
```

```
[6]: 6[2pi/8]
```

Group elements can be combined via the binary operator `@`; we can also take the inverse of an element using `~`:

```
[7]: a = G.sample()
b = G.sample()
print(a)
print(b)
print(a @ b)
print(~a)

4[2pi/8]
4[2pi/8]
0[2pi/8]
4[2pi/8]
```

Representation theory is a fundamental element in Steerable CNNs and to construct a Fourier theory over groups. In this first section, we will introduce the essential concepts.

1.1 Group Representation

A **linear group representation** ρ of a compact group G on a vector space (called *representation space*) \mathbb{R}^d is a *group homomorphism* from G to the general linear group $GL(\mathbb{R}^d)$, i.e. it is a map $\rho : G \rightarrow \mathbb{R}^{d \times d}$ such that:

$$\rho(g_1g_2) = \rho(g_1)\rho(g_2) \quad \forall g_1, g_2 \in G.$$

In other words, $\rho(g)$ is a $d \times d$ *invertible* matrix. We refer to d as the *size* of the representation.

Example: the Trivial Representation

The simplest example of *group representation* is the **trivial representation** which maps every element to $1 \in \mathbb{R}$, i.e. $\rho : g \mapsto 1$. One can verify that it satisfies the condition above. We can construct this representation as follows:

```
[8]: rho = G.trivial_representation
```

`rho` is an instance of `escnn.group.Representation`. This class provides some functionalities to work with group representations. We can also use it as a callable function to compute the representation of a group element; this will return a squared matrix as a `numpy.array`. Let verify that the trivial representation does indeed verify the condition above:

```
[9]: g1 = G.sample()
g2 = G.sample()
print(rho(g1) @ rho(g2))
print(rho(g1 @ g2))

[[1.]]
[[1.]]
```

Note that the trivial representation has size 1:

```
[10]: rho.size
```

```
[10]: 1
```

Example: rotations

Another common example of group representations is given by 2D rotations. Let $SO(2)$ be the group of all planar rotations; note that we can identify each rotation by an angle $\theta \in [0, 2\pi)$. Then, the standard representation of planar rotations as 2×2 rotation matrices is a representation of $SO(2)$:

$$\rho : r_\theta \mapsto \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

where $r_\theta \in SO(2)$ is a counter-clockwise rotation by θ . Let's try to build this group and, then, verify that this is a representation:

```
[11]: G = so2_group()
rho = G.standard_representation()

g1 = G.sample()
g2 = G.sample()
print(f'g1={g1}, g2={g2}, g1 * g2 = {g1 @ g2}')
```

(continues on next page)

(continued from previous page)

```

print()
print('rho(g1) @ rho(g2)')
print(rho(g1) @ rho(g2))
print()
print('rho(g1 * g2)')
print(rho(g1 * g2))

g1=2.1475959387857695, g2=6.208573785442527, g1 * g2 = 2.07298441704871

rho(g1) @ rho(g2)
[[[-0.481 -0.877]
 [ 0.877 -0.481]]]

rho(g1 * g2)
[[[-0.481 -0.877]
 [ 0.877 -0.481]]]

```

QUESTION 1

Show that any representation $\rho : G \rightarrow \mathbb{R}^{d \times d}$ also satisfies the following two properties:

- let $e \in G$ be the identity element. Then, $\rho(e)$ is the identity matrix of size d .
- let $g \in G$ and g^{-1} be its inverse (i.e. $g \cdot g^{-1} = e$). Then, $\rho(g^{-1}) = \rho(g)^{-1}$.

ANSWER 1

First question. First, note that for any $g \in G$:

$$\rho(g) = \rho(g \cdot e) = \rho(g)\rho(e)$$

Because $\rho(g)$ is invertible, we can left multiply by $\rho(g)^{-1}$ to find that $\rho(e)$ is the identity.

Second question. Note that

$$\rho(e) = \rho(g \cdot g^{-1}) = \rho(g)\rho(g^{-1})$$

Using the fact $\rho(e)$ is the identity, by left-multiplying by $\rho(g)^{-1}$ we recover the original statement.

Direct Sum

We can combine representations to build a larger representation via the **direct sum**.

Given representations $\rho_1 : G \rightarrow \mathbb{R}^{d_1 \times d_1}$ and $\rho_2 : G \rightarrow \mathbb{R}^{d_2 \times d_2}$, their *direct sum* $\rho_1 \oplus \rho_2 : G \rightarrow \mathbb{R}^{(d_1+d_2) \times (d_1+d_2)}$ is defined as

$$(\rho_1 \oplus \rho_2)(g) = \begin{bmatrix} \rho_1(g) & 0 \\ 0 & \rho_2(g) \end{bmatrix}$$

Its action is therefore given by the independent actions of ρ_1 and ρ_2 on the orthogonal subspaces \mathbb{R}^{d_1} and \mathbb{R}^{d_2} of $\mathbb{R}^{d_1+d_2}$.

Let's see an example:

```
[12]: rho_sum = rho + rho

g = G.sample()
print(rho(g))
print()
print(rho_sum(g))

[[[-0.625  0.781
  [-0.781 -0.625]

[[-0.625  0.781  0.      0.
 [-0.781 -0.625  0.      0.
 [ 0.       0.     -0.625  0.781]
 [ 0.       0.     -0.781 -0.625]]
```

Note that the direct sum of two representations has size equal to the sum of their sizes:

```
[13]: rho.size, rho_sum.size
[13]: (2, 4)
```

We can combine arbitrary many representations in this way, e.g. $\rho \oplus \rho \oplus \rho \oplus \rho$:

```
[14]: rho_sum = rho + rho + rho + rho

# or, more simply:
rho_sum = directsum([rho, rho, rho, rho])
rho_sum.size
[14]: 8
```

The Regular Representation

Another important representation is the **regular representation**. The regular representation describes the action of a group G on the vector space of functions over the group G . Assume for the moment that the group G is *finite*, i.e. $|G| < \infty$.

The set of functions over G is equivalent to the vector space $\mathbb{R}^{|G|}$. We can indeed interpret a vector $\mathbf{f} \in \mathbb{R}^{|G|}$ as a function over G , where the i -th entry of \mathbf{f} is interpreted as the value of the function on the i -th element $g_i \in G$.

The **regular representation** of G is a $|G|$ dimensional representation. Recall the left action of G on a function $f : G \rightarrow \mathbb{R}$:

$$[g.f](h) := f(g^{-1}h)$$

The new function $g.f$ is still a function over G and belongs to the same vector space. If we represent the function f as a vector \mathbf{f} , the vector representing the function $g.f$ will have permuted entries with respect to \mathbf{f} . This permutation is the regular representation of $g \in G$.

QUESTION 2

Show that the space of functions over G is a vector space. To do so, show that functions satisfy the properties of a vector space; see [here](#).

ANSWER 2

Let $f_1, f_2, f_3 : G \rightarrow \mathbb{R}$ be three functions and $\alpha, \beta \in \mathbb{R}$ scalars. The point-wise sum of two functions is the function $[f_1 + f_2] : G \rightarrow \mathbb{R}$ defined as

$$[f_1 + f_2](g) = f_1(g) + f_2(g)$$

The scalar multiplication is also defined pointwise as

$$[\alpha \cdot f_1](g) = \alpha f_1(g)$$

We now verify the required properties of a vector space.

- associativity: $[f_1 + (f_2 + f_3)](g) = f_1(g) + f_2(g) + f_3(g) = [(f_1 + f_2) + f_3](g)$
 - commutativity: $[f_1 + f_2](g) = f_1(g) + f_2(g) = f_2(g) + f_1(g) = [f_2 + f_1](g)$
 - identity: define the function $\mathbf{O} : G \rightarrow 0$; $[f_1 + \mathbf{O}](g) = f_1(g) + \mathbf{O}(g) = f_1(g)$
 - inverse: define $[-f_1](g) = -1 \cdot f_1(g)$; then $[f_1 + (-f_1)](g) = f_1(g) - f_1(g) = 0$
 - compatibility: $[\alpha \cdot (\beta \cdot f_1)](g) = \alpha \beta f_1(g) = [(\alpha \beta) \cdot f_1](g)$
 - identity (multiplication): $[1 \cdot f_1](g) = 1 f_1(g) = f_1(g)$
 - distributivity (vector): $[\alpha \cdot (f_1 + f_2)](g) = \alpha(f_1 + f_2)(g) = \alpha f_1(g) + \alpha f_2(g)$
 - distributivity (scalar): $[(\alpha + \beta) \cdot f_1](g) = (\alpha + \beta)f_1(g) = \alpha f_1(g) + \beta f_1(g)$
-

For finite groups, we can generate this representation. We assume that the i -th entry is associated with the element of $G = C_8$ corresponding to a rotation by $i \frac{2\pi}{8}$.

```
[15]: G = cyclic_group(8)
rho = G.regular_representation
```

```
[16]: # note that the size of the representation is equal to the group's order |G|
rho.size
```

```
[16]: 8
```

the identity element maps a function to itself, so the entries are not permuted

```
[17]: rho(G.identity)

[17]: array([[ 1.,  0., -0., -0., -0., -0.,  0., -0.],
       [ 0.,  1.,  0., -0., -0., -0., -0., -0.],
       [-0.,  0.,  1., -0., -0.,  0., -0.,  0.],
       [-0., -0., -0.,  1.,  0., -0., -0., -0.],
       [-0., -0., -0.,  0.,  1.,  0., -0., -0.],
       [-0., -0.,  0., -0.,  0.,  1., -0.,  0.],
       [ 0., -0., -0., -0., -0.,  1., -0., -0.],
       [-0., -0.,  0., -0., -0.,  0., -0.,  1.]])
```

The regular representation of the rotation by $1 \frac{2\pi}{8}$ just cyclically shifts each entry to the next position since $r_{1 \frac{2\pi}{8}}^{-1} r_i \frac{2\pi}{8} = r_{(i-1) \frac{2\pi}{8}}$:

```
[18]: rho(G.element(1))
```

```
[18]: array([[ 0., -0.,  0., -0.,  0., -0.,  1.],
       [ 1.,  0., -0., -0., -0.,  0., -0.],
       [ 0.,  1.,  0., -0.,  0., -0., -0.],
       [-0.,  0.,  1., -0., -0.,  0., -0.],
       [-0., -0., -0.,  1.,  0., -0.,  0.],
       [-0.,  0., -0.,  0.,  1.,  0., -0.],
       [-0., -0.,  0., -0.,  0.,  1.,  0.],
       [-0.,  0., -0., -0.,  0., -0.,  1.]])
```

Let's see an example of the action on a function. We consider a function which is zero on all group elements apart from the identity ($i = 0$).

```
[19]: f = np.zeros(8)
f[0] = 1
f
```

```
[19]: array([1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

Observe that $\rho(e)f = f$, where $e = 0 \frac{2\pi}{8}$ is the identity element.

```
[20]: rho(G.identity) @ f
```

```
[20]: array([ 1.,  0., -0., -0., -0., -0.,  0., -0.])
```

f is non-zero only on the element e . If an element g acts on this function, it moves the non-zero value to the entry associated with g :

```
[21]: rho(G.element(1)) @ f
```

```
[21]: array([ 0.,  1.,  0., -0., -0., -0., -0., -0.])
```

```
[22]: rho(G.element(6)) @ f
```

```
[22]: array([ 0., -0., -0., -0., -0., -0.,  1., -0.])
```

QUESTION 3

Prove the result above.

ANSWER 3

Let's call $\delta_g : G \rightarrow \mathbb{R}$ the function defined as

$$\delta_g(h) = \begin{cases} 1 & \text{if } h = g \\ 0 & \text{otherwise} \end{cases}$$

which is zero everywhere apart from $g \in G$, where it is 1. The function δ_e is represented by the vector f above.

We now want to show that $[g.\delta_e](h) = \delta_g(h)$:

$$[g.\delta_e](h) = \delta_e(g^{-1}h) = \begin{cases} 1 & \text{if } g^{-1}h = e \\ 0 & \text{otherwise} \end{cases} = \begin{cases} 1 & \text{if } h = g \\ 0 & \text{otherwise} \end{cases} = \delta_g(h)$$

Equivalent Representations

Two representations ρ and ρ' of a group G on the same vector space \mathbb{R}^d are called *equivalent* (or **isomorphic**) if and only if they are related by a change of basis $Q \in \mathbb{R}^{d \times d}$, i.e.

$$\forall g \in G \quad \rho(g) = Q\rho'(g)Q^{-1}.$$

Equivalent representations behave similarly since their composition is *basis-independent* as seen by

$$\rho'(g_1)\rho'(g_2) = Q\rho(g_1)Q^1 Q\rho(g_2)Q^1 = Q\rho(g_1)\rho(g_2)Q^1.$$

Direct sum and *change of basis matrices* provide a way to combine representations to construct larger and more complex representations. In the next example, we concatenate two trivial representations and two regular representations and apply a random change of basis Q . The final representation is formally defined as:

$$\rho(g) = Q (\rho_{\text{trivial}} \oplus \rho_{\text{regular}} \oplus \rho_{\text{regular}} \oplus \rho_{\text{trivial}}) Q^{-1}$$

```
[23]: d = G.trivial_representation.size * 2 + G.regular_representation.size * 2
Q = np.random.randn(d, d)
rho = directsum(
    [G.trivial_representation, G.regular_representation, G.regular_representation, G.
     trivial_representation],
    change_of_basis=Q
)
```

```
[24]: rho.size
```

```
[24]: 18
```

Irreducible Representations (or *Irreps*)

Under minor conditions, any representation can be decomposed in this way, that is, any representation ρ of a compact group G can be written as a *direct sum* of a number of smaller representations, up to a *change of basis*. These “smaller representations” can not be decomposed further and play a very important role in the theory of group representations and steerable CNNs and are called **irreducible representations**, or simply **irreps**.

The set of *irreducible representations* of a group G is generally denoted as \hat{G} . We will often use the notation $\hat{G} = \{\rho_j\}_j$ to index this set.

We can access the irreps of a group via the `irrep()` method. The *trivial representation* is *always* an irreducible representation. For $G = C_8$, we access it with the index $j = 0$:

```
[25]: rho_0 = G.irrep(0)

print(rho_0 == G.trivial_representation)

rho_0(G.sample())

True
```

[25]: array([[1.]])

The next irrep $j = 1$ gives the representation of $i\frac{2\pi}{8}$ as the 2×2 rotation matrix by $\theta = i\frac{2\pi}{8}$:

```
[26]: rho = G.irrep(1)
g = G.sample()

print(g)
print()
print(rho(g))

3[2pi/8]

[[[-0.707 -0.707]
 [ 0.707 -0.707]]]
```

Irreducible representations provide the building blocks to construct any representation ρ via direct sums and change of basis, i.e:

$$\rho = Q \left(\bigoplus_{j \in \mathcal{I}} \rho_j \right) Q^{-1}$$

where \mathcal{I} is an index set (possibly with repetitions) over \hat{G} .

Internally, any `escnn.group.Representation` is indeed implemented as a list of irreps (representing the index set \mathcal{I}) and a change of basis Q . An irrep is identified by a *tuple id*.

Let's see an example. Let's take the regular representaiton of C_8 and check its decomposition into irreps:

```
[27]: rho = G.regular_representation
rho.irreps
```

[27]: [(0,), (1,), (2,), (3,), (4,)]

```
[28]: rho.change_of_basis
```

[28]: array([[0.354, 0.5 , 0. , 0.5 , 0. , 0.5 , 0. , 0.354],
 [0.354, 0.354, 0.354, 0. , 0.5 , -0.354, 0.354, -0.354],
 [0.354, 0. , 0.5 , -0.5 , 0. , -0. , -0.5 , 0.354],
 [0.354, -0.354, 0.354, -0. , -0.5 , 0.354, 0.354, -0.354],
 [0.354, -0.5 , 0. , 0.5 , -0. , -0.5 , 0. , 0.354],
 [0.354, -0.354, -0.354, 0. , 0.5 , 0.354, -0.354, -0.354],
 [0.354, -0. , -0.5 , -0.5 , 0. , 0. , 0.5 , 0.354],
 [0.354, 0.354, -0.354, -0. , -0.5 , -0.354, -0.354, -0.354]])

```
[29]: # let's access second irrep
rho_id = rho.irreps[1]
rho_1 = G.irrep(*rho_id)

# we verify it is the irrep j=1 we described before
rho_1(g)

[29]: array([[-0.707, -0.707],
           [ 0.707, -0.707]])
```

Finally, let's verify that this direct sum and this change of basis indeed yield the regular representation

```
[30]: # evaluate all the irreps in rho.irreps:
irreps = [
    G.irrep(*irrep)(g) for irrep in rho.irreps
]

# build the direct sum
direct_sum = np.asarray(scipy.sparse.block_diag(irreps, format='csc')).todense()

print('Regular representation of', g)
print(rho(g))
print()
print('Direct sum of the irreps:')
print(direct_sum)
print()
print('Apply the change of basis on the direct sum of the irreps:')
print(rho.change_of_basis @ direct_sum @ rho.change_of_basis_inv)
print()
print('Are the two representations equal?', np.allclose(rho(g), rho.change_of_basis @_
      direct_sum @ rho.change_of_basis_inv))

Regular representation of 3[2pi/8]
[[ -0. -0. -0. -0. -0.  1.  0.  0.]
 [-0.  0. -0. -0.  0. -0.  1.  0.]
 [ 0. -0.  0. -0. -0.  0. -0.  1.]
 [ 1.  0. -0. -0. -0. -0.  0. -0.]
 [ 0.  1.  0. -0.  0. -0. -0. -0.]
 [-0.  0.  1. -0.  0. -0. -0.  0.]
 [ 0. -0.  0.  1.  0. -0. -0. -0.]
 [-0. -0.  0. -0.  1.  0. -0. -0.]]]

Direct sum of the irreps:
[[ 1.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.  -0.707 -0.707  0.   0.   0.   0.   0. ]
 [ 0.   0.707 -0.707  0.   0.   0.   0.   0. ]
 [ 0.   0.     0.   -0.   1.   0.   0.   0. ]
 [ 0.   0.     0.   -1.  -0.   0.   0.   0. ]
 [ 0.   0.     0.    0.   0.   0.707 -0.707  0. ]
 [ 0.   0.     0.    0.   0.   0.707  0.707  0. ]
 [ 0.   0.     0.    0.   0.   0.     0.   -1. ]]
```

(continues on next page)

(continued from previous page)

Apply the change of basis on the direct sum of the irreps:

```
[[[-0. -0. -0. -0. -0.  1.  0.  0.]
 [-0. -0. -0. -0.  0. -0.  1.  0.]
 [ 0. -0.  0. -0. -0.  0. -0.  1.]
 [ 1.  0. -0. -0. -0. -0.  0. -0.]
 [ 0.  1.  0. -0.  0. -0. -0. -0.]
 [-0.  0.  1. -0.  0. -0. -0.  0.]
 [ 0. -0.  0.  1.  0. -0. -0. -0.]
 [-0. -0.  0. -0.  1.  0. -0. -0.]]]
```

Are the two representations equal? True

1.2 Fourier Transform

We can finally approach the harmonic analysis of functions over a group G .

Note that a representation $\rho : G \rightarrow \mathbb{R}^{d \times d}$ can be interpreted as a collection of d^2 functions over G , one for each matrix entry of ρ . The **Peter-Weyl theorem** states that the collection of functions in the matrix entries of all irreps \hat{G} of a group G spans the space of all (square-integrable) functions over G .

This result gives us a way to parameterize functions over the group. This is the focus of this section. In particular, this is useful to parameterize functions over groups with infinite elements.

In this section, we will first consider the *dihedral group* D_8 as example. This is the group containing the 8 planar rotations by angles multiple of $\frac{2\pi}{8}$ and *reflection* along the X axis. The group contains in total 16 elements (8 normal rotations and 8 rotations preceded by the reflection).

```
[31]: G = dihedral_group(8)
G.order()
[31]: 16

[32]: # element representing the reflection (-) and no rotations
G.reflection
[32]: (-, 0[2pi/8])

[33]: # element representing a rotation by pi/2 (i.e. 2 * 2pi/8) and no reflections (+)
G.element((0, 2))
[33]: (+, 2[2pi/8])

[34]: # reflection followed by a rotation by pi/2
print(G.element((0, 2)) @ G.reflection)

# we can also directly generate this element as
print(G.element((1, 2)))
(-, 2[2pi/8])
(-, 2[2pi/8])

[35]: # a rotation by pi/2 followed by a reflection is equivalent to a reflection followed by ↵
# a rotation by 6*2pi/8
G.reflection @ G.element((0, 2))
```

[35]: `(-, 6[2pi/8])`

The list of all elements in the group is obtained as:

[36]: `G.elements`

[36]: `[(+, 0[2pi/8]),
(+, 1[2pi/8]),
(+, 2[2pi/8]),
(+, 3[2pi/8]),
(+, 4[2pi/8]),
(+, 5[2pi/8]),
(+, 6[2pi/8]),
(+, 7[2pi/8]),
(-, 0[2pi/8]),
(-, 1[2pi/8]),
(-, 2[2pi/8]),
(-, 3[2pi/8]),
(-, 4[2pi/8]),
(-, 5[2pi/8]),
(-, 6[2pi/8]),
(-, 7[2pi/8])]`

Fourier and Inverse Fourier Transform

For most groups, the entries of the irreps don't only span the space of functions but form also a basis (i.e. these functions are mutually orthogonal to each other). Therefore, we can write a function $f : G \rightarrow \mathbb{R}$ as

$$f(g) = \sum_{\rho_j \in \hat{G}} \sum_{m,n < d_j} w_{j,m,n} \cdot \sqrt{d_j} [\rho_j(g)]_{mn}$$

where d_j is the dimension of the irrep ρ_j , while m, n index the d_j^2 entries of ρ_j . The coefficients $\{w_{j,m,n} \in \mathbb{R}\}_{j,m,n}$ parameterize the function f on this basis. The $\sqrt{d_j}$ is a scalar factor to ensure the basis is normalized.

We rewrite this expression in a cleaner form by using the following fact. If $A, B \in \mathbb{R}^{d \times d}$, then

$$\text{Tr}(A^T B) = \sum_{m,n < d} A_{mn} B_{mn} \in \mathbb{R}.$$

By defining $\hat{f}(\rho_j) \in \mathbb{R}^{d_j \times d_j}$ as the matrix containing the d_j^2 coefficients $\{w_{j,m,n} \in \mathbb{R}\}_{m,n < d_j}$, we can express the **Inverse Fourier Transform** as:

$$f(g) = \sum_{\rho_j \in \hat{G}} \sqrt{d_j} \text{Tr} \left(\rho_j(g)^T \hat{f}(\rho_j) \right)$$

Similarly, we can project a general function $f : G \rightarrow \mathbb{R}$ on an element $\rho_{j,m,n} : G \rightarrow \mathbb{R}$ of the basis via:

$$w_{j,m,n} = \frac{1}{|G|} \sum_{g \in G} f(g) \sqrt{d_j} [\rho_j(g)]_{m,n}.$$

The projection over all entries of ρ_j can be more cleanly written as follows:

$$\hat{f}(\rho_j) = \frac{1}{|G|} \sum_{g \in G} f(g) \sqrt{d_j} \rho_j(g).$$

which we refer to as **Fourier Transform**.

If the group G is *infinite*, we replace the average over the group elements with an *integral* over them:

$$\hat{f}(\rho_j) = \int_G f(g) \sqrt{d_j} \rho_j(g) dg ,$$

For a finite group G , we can access all its irreps by using the `Group.irreps()` method. Let's see an example:

```
[37]: irreps = G.irreps()
print(f'The dihedral group D8 has {len(irreps)} irreps')

The dihedral group D8 has 7 irreps
```

```
[38]: # the first one, is the 1-dimensional trivial representation
print(irreps[0] == G.trivial_representation == G.irrep(0, 0))

True
```

QUESTION 4

We can now implement the Fourier Transform and the Inverse Fourier Transform for the Dihedral Group D_8 . Using the equations above, implement the following methods:

```
[39]: def fourier_transform_D8(f: np.array):
    # the method gets in input a function on the elements of D_8
    # and should return a dictionary mapping each irrep's `id` to the corresponding
    # Fourier Transform
    # The i-th element of `f` stores the value of the function on the group element `G`.
    # elements[i]

    G = dihedral_group(8)
    assert f.shape == (16,), f.shape
    ft = {}

    #####
    # INSERT YOUR CODE HERE:

    for rho in G.irreps():
        d = rho.size

        rho_g = np.stack([rho(g) for g in G.elements], axis=0)

        ft[rho.id] = (f.reshape(-1, 1, 1) * rho_g).mean(0) * np.sqrt(d)

    #####
    return ft
```

```
[40]: def inverse_fourier_transform_D8(ft: dict):
    # the method gets in input a dictionary mapping each irrep's `id` to the
    # corresponding Fourier Transform
    # and should return the function `f` on the elements of D_8
    # The i-th element of `f` stores the value of the function on the group element `G.
    # elements[i]`

    G = dihedral_group(8)
    f = np.zeros(16)

    #####
    # INSERT YOUR CODE HERE:
    for rho in G.irreps():
        d = rho.size
        for i, g in enumerate(G.elements):
            f[i] += np.sqrt(d) * (ft[rho.id] * rho(g)).sum()
    ####

    return f
```

We now want to verify that the **Fourier Transform** and the **Inverse Fourier Transform** are inverse of each other:

```
[41]: f = np.random.randn(16)

ft = fourier_transform_D8(f)

new_f = inverse_fourier_transform_D8(ft)

assert np.allclose(f, new_f)
```

Parameterizing functions over infinite groups

This allows us to also parameterize functions over infinite groups, such as $O(2)$, i.e. the group of all planar rotations and reflections.

```
[42]: G = o2_group()
```

```
[43]: # the group has infinite many elements, so the `order` method just returns -1
G.order()
```

```
[43]: -1
```

The equations remain the same, but this group has an *infinite* number of *irreps*. We can, however, parameterize a function over the group by only considering a finite number of irreps in the sum inside the definition of *Inverse Fourier Transform*. Let $\tilde{G} \subset \hat{G}$ be a finite subset of the irreps of G . We can then write the following transforms within the subspace of functions spanned only by the entries of the irreps in \tilde{G} .

Inverse Fourier Transform:

$$f(g) = \sum_{\rho_j \in \tilde{G}} \sqrt{d_j} \text{Tr} \left(\rho_j(g)^T \hat{f}(\rho_j) \right)$$

and Fourier Transform:

$$\hat{f}(\rho_j) = \int_G f(g) \sqrt{d_j} \rho_j(g) dg ,$$

QUESTION 5

We can now implement the Inverse Fourier Transform for the Orthogonal Group $O(2)$. Since the group has infinite many elements, we can not store the values the function take on each element. Instead, we just sample the function on a particular element of the group:

```
[44]: def inverse_fourier_transform_O2(g: GroupElement, ft: dict):
    # the method gets in input a dictionary mapping each irrep's `id` to the
    # corresponding Fourier Transform
    # and a group element `g`
    # The method should return the value of the function evaluated on `g`.

    G = o2_group()
    f = 0

    #####
    # INSERT YOUR CODE HERE:
    for rho, ft_rho in ft.items():
        rho = G.irrep(*rho)
        d = rho.size
        f += np.sqrt(d) * (ft_rho * rho(g)).sum()
    #####
    return f
```

Let's plot a function. First we generate a random function by using a few irreps.

```
[45]: irreps = [G.irrep(0, 0)] + [G.irrep(1, j) for j in range(3)]

ft = {
    rho.id: np.random.randn(rho.size, rho.size)
    for rho in irreps
}
```

Then, we generate a grid on the group where to evaluate the function, i.e. we choose a finite set of element of G . Like the Dihedral group, $O(2)$ contains rotations (parameterized by an angle $\theta \in [0, 2\pi)$) and a reflection followed by any rotation. For example:

```
[46]: G.sample()

[46]: (-, 5.686258713110608)
```

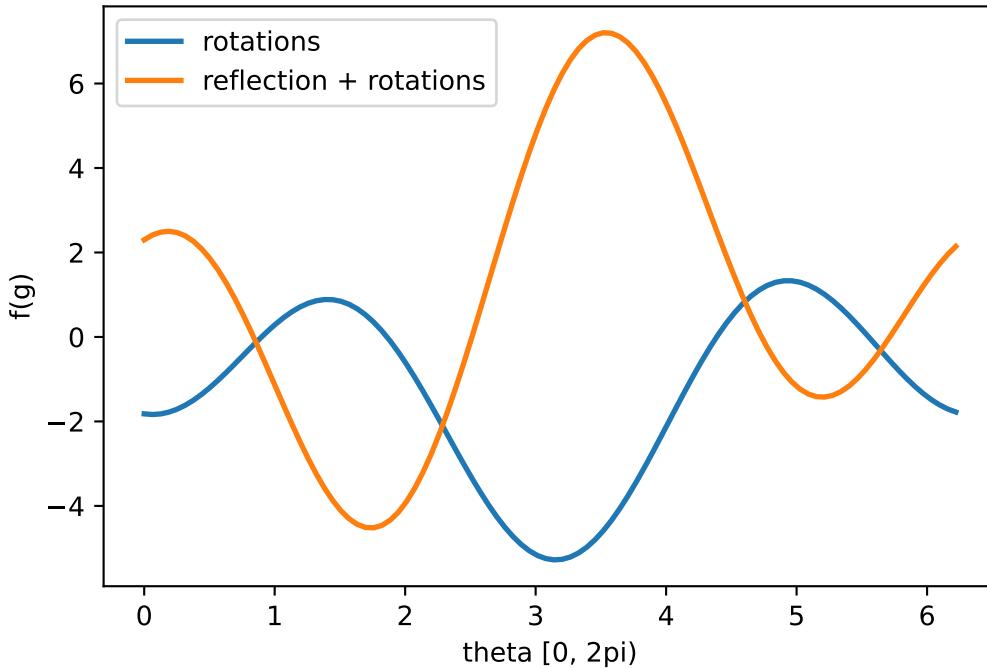
To build our grid, we sample 100 rotations and 100 rotations preceeded by a reflection:

```
[47]: N = 100
thetas = [i*2*np.pi/N for i in range(N)]
grid_rot = [G.element((0, theta)) for theta in thetas]
grid_refl = [G.element((1, theta)) for theta in thetas]
```

We now evaluate the function over all these elements and, finally, plot it:

```
[48]: f_rot = [
    inverse_fourier_transform_02(g, ft) for g in grid_rot
]
f_refl = [
    inverse_fourier_transform_02(g, ft) for g in grid_refl
]

plt.plot(thetas, f_rot, label='rotations')
plt.plot(thetas, f_refl, label='reflection + rotations')
plt.xlabel('theta [0, 2pi]')
plt.ylabel('f(g)')
plt.legend()
plt.show()
```



Observe that using more irreps allows one to parameterize more flexible functions. Let's try to add some more:

```
[49]: irreps = [G.irrep(0, 0)] + [G.irrep(1, j) for j in range(8)]

ft = {
    rho.id: np.random.randn(rho.size, rho.size)
    for rho in irreps
}
```

(continues on next page)

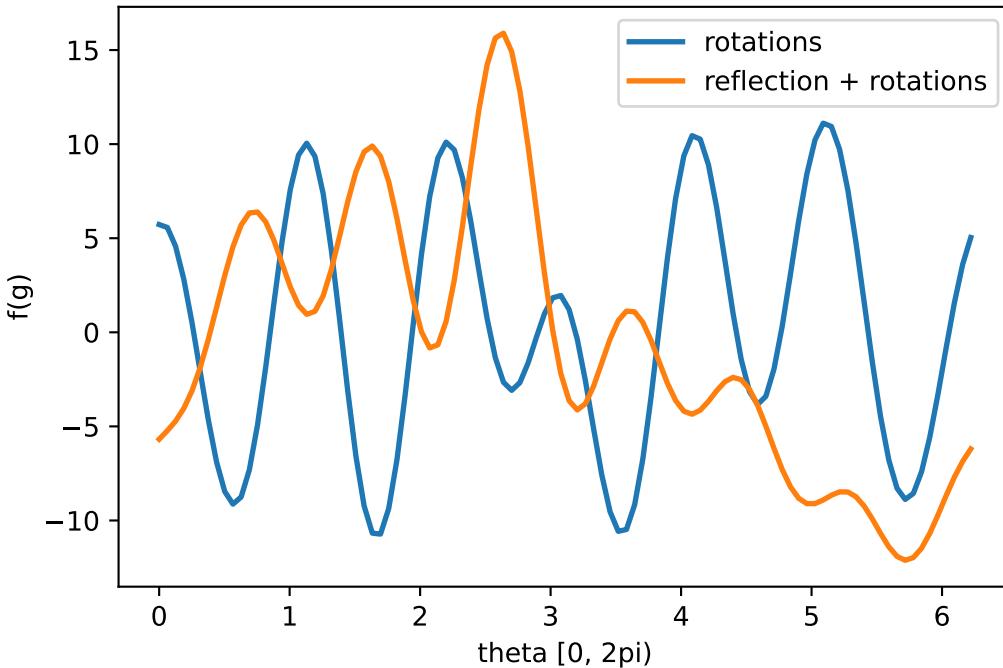
(continued from previous page)

```

f_rot = [
    inverse_fourier_transform_02(g, ft) for g in grid_rot
]
f_refl = [
    inverse_fourier_transform_02(g, ft) for g in grid_refl
]

plt.plot(thetas, f_rot, label='rotations')
plt.plot(thetas, f_refl, label='reflection + rotations')
plt.xlabel('theta [0, 2pi]')
plt.ylabel('f(g)')
plt.legend()
plt.show()

```



Fourier Transform of shifted functions

Recall that a group element $g \in G$ can act on a function $f : G \rightarrow \mathbb{R}$ as:

$$[g.f](h) = f(g^{-1}h).$$

The Fourier transform defined before has the convenient property that the Fourier transform of f and of $[g.f]$ are related as follows:

$$\widehat{g.f}(\rho_j) = \rho_j(g)\widehat{f}$$

for any irrep ρ_j .

QUESTION 6

Prove the property above.

ANSWER 6

$$\widehat{g \cdot f}(\rho_j) = \int_G [g \cdot f](h) \sqrt{d_j} \rho_j(h) dh \quad (4.1)$$

$$= \int_G f(g^{-1}h) \sqrt{d_j} \rho_j(h) dh$$

Define $t = g^{-1}h$ and, therefore, $h = g t$

$$= \int_G f(t) \sqrt{d_j} \rho_j(gt) dt$$

$$= \int_G f(t) \sqrt{d_j} \rho_j(g) \rho_j(t) dt$$

$$= \rho_j(g) \int_G f(t) \sqrt{d_j} \rho_j(t) dt$$

$$= \rho_j(g) \widehat{f}(\rho_j)$$

We can verify this property visually:

```
[50]: irreps = [G.irrep(0, 0)] + [G.irrep(1, j) for j in range(8)]

# first, we generate a random function, as earlier
ft = {
    rho.id: np.random.randn(rho.size, rho.size)
    for rho in irreps
}

# second, we sample a random group element `g`
g = G.sample()
print(f'Transforming the function with g={g}')

# finally, we transform the Fourier coefficients as in the equations above:
gft = {
    rho.id: rho(g) @ ft[rho.id]
    for rho in irreps
}

# Let's now visualize the two functions:

f_rot = [
    inverse_fourier_transform_02(g, ft) for g in grid_rot
]
f_refl = [
    inverse_fourier_transform_02(g, ft) for g in grid_refl
]
```

(continues on next page)

(continued from previous page)

```

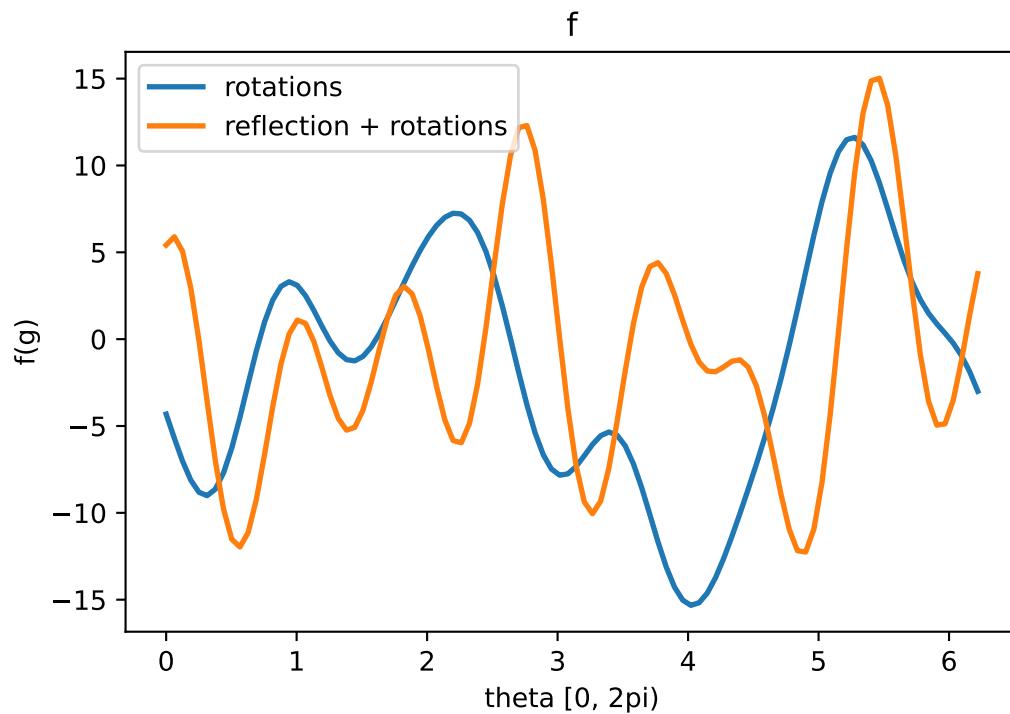
gf_rot = [
    inverse_fourier_transform_02(g, gft) for g in grid_rot
]
gf_refl = [
    inverse_fourier_transform_02(g, gft) for g in grid_refl
]

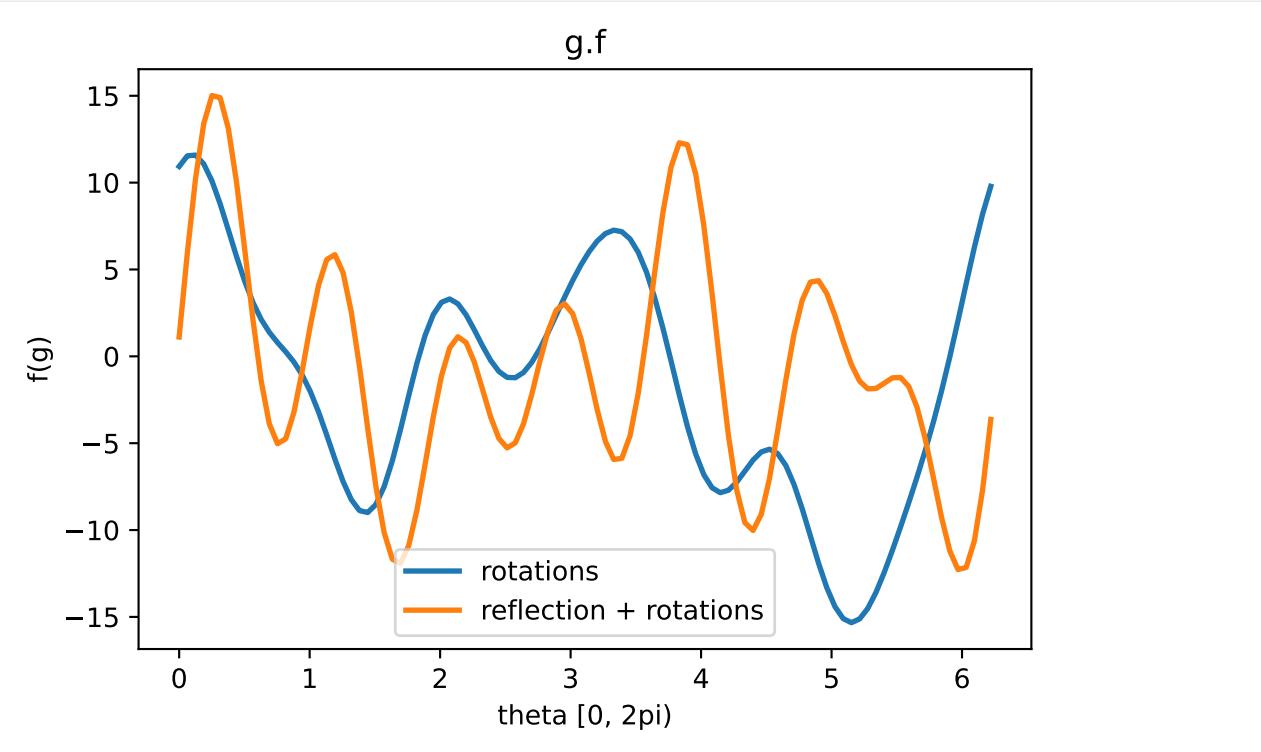
plt.plot(thetas, f_rot, label='rotations')
plt.plot(thetas, f_refl, label='reflection + rotations')
plt.xlabel('theta [0, 2pi]')
plt.ylabel('f(g)')
plt.title('f')
plt.legend()
plt.show()

plt.plot(thetas, gf_rot, label='rotations')
plt.plot(thetas, gf_refl, label='reflection + rotations')
plt.xlabel('theta [0, 2pi]')
plt.ylabel('f(g)')
plt.title('g.f')
plt.legend()
plt.show()

```

Transforming the function with $g=(+, 1.1212128744532228)$





From the Fourier Transform to the Regular Representation

For simplicity, we can stack all the Fourier coefficients (the output of the Fourier transform, that is, the input of the inverse Fourier transform) into a unique vector. We define the vector \mathbf{f} as the stack of the columns of each Fourier coefficients matrix $f(\rho_j)$.

Let's first introduce some notation. We denote the stack of two vectors $\mathbf{v}_1, \mathbf{v}_2$ as $\mathbf{v}_1 \oplus \mathbf{v}_2$. The use of \oplus is not random: if ρ_1 is a representation acting on \mathbf{v}_1 and ρ_2 is a representation acting on \mathbf{v}_2 , then the *direct sum* $\rho_1 \oplus \rho_2$ acts on the concatenated vector $\mathbf{v}_1 \oplus \mathbf{v}_2$.

Second, we denote by $\text{vec}(A)$ the vector which is the stack of the columns of a matrix A . In `numpy`, this is written as `A.T.reshape(-1)`, where the transpose is necessary since `numpy` stacks rows by default.

Then, we write:

$$\mathbf{f} = \bigoplus_{\rho_j} \text{vec}(\hat{f}(\rho_j)) .$$

Moreover, by using $\widehat{g.f}(\rho_j) = \rho_j(g)\hat{f}(\rho_j)$, we see that the vector containing the coefficients of the function $[g.f]$ will be:

$$\bigoplus_{\rho_j} \text{vec}(\rho_j(g)\hat{f}(\rho_j)) = \bigoplus_{\rho_j} \left(\bigoplus_{j=1}^{d_j} \rho_j(g) \right) \text{vec}(\hat{f}(\rho_j))$$

In other words, the group G is acting on the vector \mathbf{f} with the following representation:

$$\rho(g) = \bigoplus_{\rho_j} \bigoplus_{j=1}^{d_j} \rho_j(g)$$

i.e. $\rho(g)\mathbf{f}$ is the vector containing the Fourier coefficients of the function $[g.f]$.

Note that, essentially, the representation ρ acts on a vector space containing functions over G . This should remind you of the **regular representation** we defined for *finite groups* earlier. Indeed, it turns out that, if G is finite, the representation ρ we have just constructed is **isomorphic** (*equivalent*) to the *regular representation* defined earlier. The change of basis Q is a matrix which performs the Fourier transform, while Q^{-1} performs the inverse Fourier transform. More formally:

$$\rho_{\text{reg}}(g) = Q^{-1} \left(\bigoplus_{\rho_j} \bigoplus_{j=1}^{d_j} \rho_j(g) \right) Q$$

where each irrep ρ_j is repeated d_j times, i.e. a number of times equal to its size.

Intuition: recall that a function $f : G \rightarrow \mathbb{R}$ is just a vector living in a vector space. Such vector can be expressed with respect to any basis for this vector space. The first time we introduced the *regular representation* for finite groups, we chose a basis where each axis is associated with a group element; the action of G is realized in this basis by a permutation of all the axes. Here, instead, we defined a basis for the same vector space where G acts independently on different subsets of the axes, i.e. the action of G is a block-diagonal matrix (the direct sum of irreps). This is often a more convenient choice of basis as we will see later.

Let verify this equivalence for the Dihedral group D_8 :

```
[51]: G = dihedral_group(8)

rho_irreps = []
for rho_j in G.irreps():
    d_j = rho_j.size
    # repeat each irrep a number of times equal to its size
    rho_irreps += [rho_j]*d_j

rho = directsum(rho_irreps)

print('The representations have the same size:')
print(rho.size, G.regular_representation.size)

print('And contain the same irreps:')
print(rho.irreps)
print(G.regular_representation.irreps)

# Fourier transform matrix:
Q = G.regular_representation.change_of_basis
# inverse Fourier transform matrix:
Qinv = G.regular_representation.change_of_basis_inv

# let's check that the two representations are indeed equivalent
g = G.sample()

rho_g = rho(g)
reg_g = G.regular_representation(g)
print()
print('Are the two representations equivalent?', np.allclose(Q @ rho_g @ Qinv, reg_g))

The representations have the same size:
16 16
```

(continues on next page)

(continued from previous page)

And contain the same irreps:

```
[(0, 0), (1, 0), (1, 1), (1, 1), (1, 2), (1, 2), (1, 3), (1, 3), (1, 4), (0, 4)]
[(0, 0), (1, 0), (1, 1), (1, 1), (1, 2), (1, 2), (1, 3), (1, 3), (1, 4), (0, 4)]
```

Are the two representations equivalent? True

When G is not finite, we can not explicitly store the regular representation ρ_{reg} or the Fourier transform matrix Q , since they are infinite dimensional. Nevertheless, as we have done earlier, we can just consider a subset of all functions, spanned only by a finite number of irreps. We can sample the function on any group element via the Inverse Fourier Transform when needed, without the need to compute the full Inverse Fourier Transform Q^{-1} to store all values.

This is the underlying idea we will exploit later to build GCNNs equivariant to infinite groups.

We can easily generate this representation as:

```
[52]: G = o2_group()
irreps = [G.irrep(0, 0)] + [G.irrep(1, j) for j in range(8)]
rho = G.spectral_regular_representation(*irreps, name='regular_representation')
```

`rho.irreps`

```
[52]: [(0, 0),
        (1, 0),
        (1, 1),
        (1, 1),
        (1, 2),
        (1, 2),
        (1, 3),
        (1, 3),
        (1, 4),
        (1, 4),
        (1, 5),
        (1, 5),
        (1, 6),
        (1, 6),
        (1, 7),
        (1, 7)]
```

Irreps with redundant entries: the case of $SO(2)$

We need to conclude with a final note about the Fourier transform. When we introduced it earlier, we said that the entries of the irreps form a **basis** for the functions over *most* groups. Indeed, there exists some groups where the entries of the irreps are partially redundant and, therefore, form an *overcomplete* basis. This is the case, for example, of the group of planar rotations $SO(2)$ (or the group of N discrete rotations C_N). Indeed, an irrep of $SO(2)$ has form:

$$\rho_j(r_\theta) = \begin{bmatrix} \cos(j \cdot \theta) & -\sin(j \cdot \theta) \\ \sin(j \cdot \theta) & \cos(j \cdot \theta) \end{bmatrix}$$

for $\theta \in [0, 2\pi]$, where the integer $j \in \mathbb{N}$ is interpreted as the rotational *frequency*.

You can observe that the two columns of $\rho_j(r_\theta)$ contain redundant elements and span the same 2 dimensional space of functions. It is indeed sufficient to consider only one of the two columns to parameterize functions over $SO(2)$. This also means that the irrep ρ_j appears only once (instead of $d_j = 2$ times) in the regular representation.

We don't generally need to worry much about this, since we can generate the representation as earlier:

```
[53]: G = so2_group()
irreps = [G.irrep(j) for j in range(8)]
rho = G.spectral_regular_representation(*irreps, name='regular_representation')

# observe that each irrep is now repeated only once, even if some are 2-dimensional
rho.irreps

[53]: [(0,), (1,), (2,), (3,), (4,), (5,), (6,), (7,)]
```

4.31.3 2. From Group CNNs to Steerable CNNs

We consider a GCNN equivariant to a *semi-direct* product group $\mathbb{R}^n \rtimes G$, with compact group $G \leq O(n)$. This setting covers equivariance to **isometries** (distance preserving transformations) of the Euclidean space \mathbb{R}^n ; in particular, it includes equivariance to *translations* in \mathbb{R}^n and to a origin-preserving symmetry G (e.g. rotations or reflections in n -dimensions). We call G a **point group**.

If $G = O(n)$, the group of all rotations and reflections in \mathbb{R}^n , then $E(n) = \mathbb{R}^n \rtimes O(n)$ is called the **Euclidean group**, and includes all isometries of \mathbb{R}^n .

2.1 Feature Fields

In a GCNN, a feature map is a signal $f : \mathbb{R}^n \times G \rightarrow \mathbb{R}$. The action of an element $(x, g) \in \mathbb{R}^n \rtimes G$ is:

$$[(x, g).f](y, h) := f(g^{-1}(y - x), g^{-1}h)$$

where $x, y \in \mathbb{R}^n$ and $g, h \in G$.

QUESTION 7

Prove the action has indeed this form.

ANSWER 7

First, recall the group law: for any (x, g) and $(y, h) \in \mathbb{R}^n \rtimes G$

$$(x, g) \cdot (y, h) = (x + g.y, gh)$$

where $x, y, g.y \in \mathbb{R}^n$ and $g, h \in G$. Second, recall the inverse element is $(x, g)^{-1} = (-g^{-1}.x, g^{-1})$. Then:

$$[(x, g).f](y, h) = f((x, g)^{-1} \cdot (y, h)) = f(-g^{-1}.x + g^{-1}.y, g^{-1}h) = f(g^{-1}.(y - x), g^{-1}h)$$

In a GCNN, a feature map f is stored as a multi-dimensional array with an axis for each of the n spatial dimensions and one for the group G .

In a steerable CNN, we replace the G axis with a “Fourier” axis, which contains c Fourier coefficients used to parameterize a function over G , as described in the previous section. Again, let’s call $\rho : G \rightarrow \mathbb{R}^{c \times c}$ the representation of G acting on these c coefficients. The result is equivalent to a standard GCNN if G is finite (and we have $c = |G|$), but we can now also use infinite G , such as $SO(2)$.

A feature map f can now be interpreted as a vector field on the space \mathbb{R}^n , i.e.:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^c$$

which assigns a c -dimensional feature vector $f(x) \in \mathbb{R}^c$ to each spatial position $x \in \mathbb{R}^n$. We call such vector field a **feature vector field**.

The action of $\mathbb{R}^n \rtimes G$ on one such feature vector field is defined as:

$$[(x, g).f](y) := \rho(g)f(g^{-1}(y - x))$$

where $x, y \in \mathbb{R}^n$ and $g \in G$.

QUESTION 8

Prove that this is indeed the right action of $\mathbb{R}^n \rtimes G$ on the feature vector field $f : \mathbb{R}^n \rightarrow \mathbb{R}^c$. Recall the action of this group over the functions of the form $\underline{f} : \mathbb{R}^n \rtimes G \rightarrow \mathbb{R}$ that we described earlier. Moreover, note that the vector $f(x) \in \mathbb{R}^c$ contains the c Fourier coefficients of the function $\widehat{f}(x, \cdot) : G \rightarrow \mathbb{R}$ along its G axis, i.e.:

$$f(x) = \bigoplus_{\rho_j} \text{vec} \left(\widehat{f}(x, \cdot)(\rho_j) \right)$$

ANSWER 8:

We know from the previous question that

$$[(x, g).\underline{f}](y, h) = \underline{f}(g^{-1}(y - x), g^{-1}h)$$

Recall also that $\rho(g) = \bigoplus_{\rho_j} \bigoplus^{d_j} \rho_j(g) \in \mathbb{R}^{c \times c}$ is the regular representation of G acting on the vector of Fourier coefficients. Then:

$$\begin{aligned} [(x, g).f](y) &= \bigoplus_{\rho_j} \text{vec} \left(\widehat{[(x, g).\underline{f}]}(y, \cdot) \right) (\rho_j) \\ &= \bigoplus_{\rho_j} \text{vec} \left(\widehat{\underline{f}(g^{-1}(y - x), g^{-1}\cdot)} \right) (\rho_j) \\ &= \bigoplus_{\rho_j} \text{vec} \left(\rho_j(g) \left[\widehat{f}(g^{-1}(y - x), \cdot) \right] (\rho_j) \right) \\ &= \rho(g)f(g^{-1}(y)) \end{aligned} \tag{4.8}$$

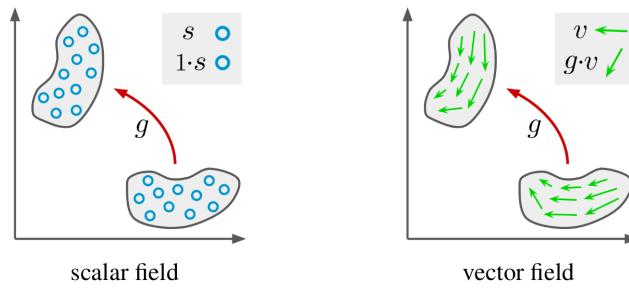
Note that in the equations above, the square brackets in $\widehat{[\cdot]}$ indicate that $\widehat{\cdot}$ covers the whole content of the brackets.

General Steerable CNNs

The framework of Steerable CNNs is actually more general and allows for any representation ρ of G . A different choice of ρ generally require some structural change in the architecture, e.g. by adapting the non-linearity used to ensure equivariance. Anyways, for simplicity, we will stick with the Fourier example in this tutorial.

Throughout the rest of this tutorial, we will assume $n = 2$ for simplicity. That means we will be working for example with planar images and with the isometries of the plane (2D rotations or mirroring). The actions of $g \in G = SO(2)$

on two examples of feature vector fields over \mathbb{R}^2 are shown next. On the left, ρ is the trivial representation of $SO(2)$ while, on the right, ρ is the representation of $SO(2)$ as 2×2 rotation matrices.



2.2 Defining a Steerable CNN

We can now proceed with building a Steerable CNN. First we import some other useful packages.

```
[54]: from escnn import group
from escnn import gspaces
from escnn import nn
```

First, we need to choose the group G of point symmetries (reflections and rotations) which are being considered. All of these choices are subgroups $G \leq O(2)$ of the orthogonal group.

For simplicity, we first consider the *finite* group $G = C_4$, which models the 4 *rotations* by angle $\theta \in \{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\}$. Because these are perfect symmetries of the grid, transforming an image with this group does not require any interpolation. We will later extend our examples to an infinite group such as $SO(2)$ or $O(2)$.

Recall that a semi-direct product $\mathbb{R}^2 \rtimes G$ is defined by G but also by the action of G on \mathbb{R}^2 . We determine both the **point group** G and its **action on the space** \mathbb{R}^2 by instantiating a subclass of `gspace.GSpace`. For the rotational action of $G = C_4$ on \mathbb{R}^2 this is done by:

```
[55]: r2_act = gspaces.rot2dOnR2(N=4)
r2_act
```

```
[55]: C4_on_R2[(None, 4)]
```

```
[56]: # we can access the group G as
G = r2_act.fibergroup
G
```

```
[56]: C4
```

Having specified the symmetry transformation on the *base space* \mathbb{R}^2 , we next need to define the representation $\rho : G \rightarrow \mathbb{R}^{c \times c}$ which describes how a **feature vector field** $f : \mathbb{R}^2 \rightarrow \mathbb{R}^c$ transforms under the action of G . This transformation law of feature fields is implemented by `nn.FieldType`.

We instantiate the `nn.FieldType` modeling a GCNN feature by passing it the `gspaces.GSpace` instance and the *regular representation* of $G = C_4$. We call a feature field associated with the regular representation ρ_{reg} a **regular feature field**.

```
[57]: feat_type = nn.FieldType(r2_act, [G.regular_representation])
feat_type
```

```
[57]: [C4_on_R2[(None, 4)]: {regular (x1)}(4)]
```

Recall that the regular representation of a finite group G built by `G.regular_representation` is a permutation matrix of shape $|G| \times |G|$:

```
[58]: G.regular_representation(G.sample())
[58]: array([[-0., -0.,  1.,  0.],
           [ 0., -0., -0.,  1.],
           [ 1.,  0., -0., -0.],
           [ 0.,  1.,  0., -0.]])
```

Deep Feature spaces

The deep feature spaces of a GCNN typically comprise multiple channels. Similarly, the feature spaces of a steerable CNN can include multiple independent feature fields. This is achieved via **direc sum**, but stacking multiple copies of ρ .

For example, we can use 3 copies of the regular representation $\rho_{\text{reg}} : G \rightarrow \mathbb{R}^{|G|}$. The full feature space is in this case modeled as a *stacked* field $f : \mathbb{R}^2 \rightarrow \mathbb{R}^{3|G|}$ which transforms according to the **direct sum** of three regular representations:

$$\rho(r_\theta) = \rho_{\text{reg}}(r_\theta) \oplus \rho_{\text{reg}}(r_\theta) \oplus \rho_{\text{reg}}(r_\theta) = \begin{bmatrix} \rho_{\text{reg}}(\theta) & 0 & 0 \\ 0 & \rho_{\text{reg}}(\theta) & 0 \\ 0 & 0 & \rho_{\text{reg}}(\theta) \end{bmatrix} \in \mathbb{R}^{3|G| \times 3|G|}$$

We instantiate a `nn.FieldType` composed of 3 regular representations by passing the full field representation as a list of three regular representations:

```
[59]: # Technically, one can also construct the direct-sum representation G.regular_
       ↪representation + G.regular_representation + G.regular_representation as done
       ↪# before. Passing a list containing 3 copies of G.regular_representation allows for more
       ↪efficient implementation of certain operations internally.
feat_type = nn.FieldType(r2_act, [G.regular_representation]*3)
feat_type
[59]: [C4_on_R2[(None, 4)]: {regular (x3)}(12)]
```

Input Features

Each hidden layer of a steerable CNN has its own transformation law which the user needs to specify (equivalent to the choice of number of channels in each layer of a conventional CNN). The *input* and *output* of a steerable CNN are also feature fields and their type (i.e. transformation law) is typically determined by the inference task.

The most common example is that of gray-scale input images. A rotation of a gray-scale image is performed by moving each pixel to a new position without changing their intensity values. The invariance of the scalar pixel values under rotations is modeled by the **trivial representation** $\rho_0 : G \rightarrow \mathbb{R}$, $g \mapsto 1$ of G and identifies them as **scalar fields**. Formally, a scalar field is a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ mapping to a feature vector with $c = 1$ channels. A rotation $r_\theta \in C_4$ transforms this scalar field as

$$[r_\theta . f](x) := \rho_0(r_\theta) f(r_\theta^{-1}x) = 1 \cdot f(r_\theta^{-1}x) = f(r_\theta^{-1}x).$$

We instantiate the `nn.FieldType` modeling a gray-scale image by passing it the trivial representation of G :

```
[60]: feat_type_in = nn.FieldType(r2_act, [G.trivial_representation])
feat_type_in
```

[60]: [C4_on_R2[(None, 4)]: {irrep_0 (x1)}(1)]

Equivariant Layers

When we build a model **equivariant** to a group G , we require that the output produced by the model transforms consistently when the input transforms under the action of an element $g \in G$. For a function F (e.g. a neural network), the **equivariance constraint** requires:

$$\mathcal{T}_g^{\text{out}}[F(x)] = F(\mathcal{T}_g^{\text{in}}[x]) \quad \forall g \in G$$

where $\mathcal{T}_g^{\text{in}}$ is the transformation of the input by the group element g while $\mathcal{T}_g^{\text{out}}$ is the transformation of the output by the same element. The *field type* `feat_type_in` we have just defined above precisely describes \mathcal{T}^{in} . The transformation law \mathcal{T}^{out} of the output of the first layer is similarly chosen by defining an instance `feat_type_out` of `nn.FieldType`.

For example, let's use 3 *regular feature fields* in output:

[61]: `feat_type_out = nn.FieldType(r2_act, [G.regular_representation]*3)`

As a shortcut, we can also use:

[62]: `feat_type_in = nn.FieldType(r2_act, [r2_act.trivial_repr])`
`feat_type_out = nn.FieldType(r2_act, [r2_act.regular_repr]*3)`

Once having defined how the input and output feature spaces should transform, we can build neural network functions as **equivariant modules**. These are implemented as subclasses of an abstract base class `nn.EquivariantModule` which itself inherits from `torch.nn.Module`.

Equivariant Convolution Layer: We start by instantiating a convolutional layer that maps between fields of types `feat_type_in` and `feat_type_out`.

Let $\rho_{\text{in}} : G \rightarrow \mathbb{R}^{c_{\text{in}} \times c_{\text{in}}}$ and $\rho_{\text{out}} : G \rightarrow \mathbb{R}^{c_{\text{out}} \times c_{\text{out}}}$ be respectively the representations of G associated with `feat_type_in` and `feat_type_out`. Then, an equivariant convolution layer is a standard convolution layer with a filter $k : \mathbb{R}^2 \rightarrow \mathbb{R}^{c_{\text{out}} \times c_{\text{in}}}$ (note the number of input and output channels) which satisfies a particular **steerability constraint**:

$$\forall g \in G, x \in \mathbb{R}^2 \quad k(g.x) = \rho_{\text{out}}(g)k(x)\rho_{\text{in}}(g)^{-1}$$

In particular, the use of convolution guarantees the translation equivariance, while the fact the filters satisfy this steerability constraint guarantees the G -equivairance.

QUESTION 9

Show that if a filter $k : \mathbb{R}^2 \rightarrow \mathbb{R}^{c_{\text{out}} \times c_{\text{in}}}$ satisfies the constraint above, the convolution with it is equivariant to G , i.e. show that

$$f_{\text{out}} = k * f_{\text{in}} \implies [g.f_{\text{out}}] = k * [g.f_{\text{in}}]$$

for all $g \in G$.

The action on the features f_{in} and f_{out} is the one previously defined, i.e:

$$[g.f_{\text{in}}](x) = \rho_{\text{in}}(g)f(g^{-1}x)$$

and

$$[g.f_{\text{out}}](x) = \rho_{\text{out}}(g)f(g^{-1}x)$$

while the convolution is defined as

$$f_{\text{out}}(y) = [k \star f_{\text{in}}](y) = \int_{\mathbb{R}^2} k(x-y)f_{\text{in}}(x)dx$$

ANSWER 9

Note that, because k satisfies the steerable constraint, it follows that $k(x) = \rho_{\text{out}}(g)k(g^{-1}.x)\rho_{\text{in}}(g)^{-1}$. Then:

$$\begin{aligned} k \star [g.f_{\text{in}}](y) &= \int_{\mathbb{R}^2} k(x-y)[g.f_{\text{in}}](x)dx \\ &= \int_{\mathbb{R}^2} k(x-y)\rho_{\text{in}}(g)f_{\text{in}}(g^{-1}(x-y))dx \\ &= \rho_{\text{out}}(g) \int_{\mathbb{R}^2} k(g^{-1}.(x-y))f_{\text{in}}(g^{-1}(x-y))dx \end{aligned} \quad (4.12)$$

Define $z = g^{-1}(x-y)$

$$\begin{aligned} &= \rho_{\text{out}}(g) \int_{\mathbb{R}^2} k(z-g^{-1}.y)f_{\text{in}}(g^{-1}(z))dz \\ &= \rho_{\text{out}}(g)f_{\text{out}}(g^{-1}y) \\ &= [g.f_{\text{out}}](y) \end{aligned}$$

The steerability constraint restricts the space of possible learnable filters to a smaller space of equivariant filters. Solving this constraint goes beyond the scope of this tutorial; fortunately, the `nn.R2Conv` module takes care of properly parameterizing the filter k such that it satisfies the constraint.

```
[63]: conv = nn.R2Conv(feat_type_in, feat_type_out, kernel_size=3)
```

Each equivariant module has an input and output type. As a function `(.forward())`, it *requires* its inputs to transform according to its input type and is guaranteed to return feature fields associated with its output type. To prevent the user from accidentally feeding an incorrectly transforming input field into an equivariant module, we perform a dynamic type checking. In order to do so, we define **geometric tensors** as data containers. They are wrapping a *PyTorch* `torch.Tensor` to augment them with an instance of `FieldType`.

Let's build a few random 32x32 gray-scale images and wrap them into an `nn.GeometricTensor`:

```
[64]: x = torch.randn(4, 1, 32, 32)
# FieldType is a callable object; its call method can be used to wrap PyTorch tensors
# into GeometricTensors
x = feat_type_in(x)

assert isinstance(x.tensor, torch.Tensor)
assert isinstance(x, nn.GeometricTensor)
```

As usually done in *PyTorch*, an image or feature map is stored in a 4-dimensional array of shape $B \times C \times H \times W$, where B is the batch-size, C is the number of channels and W and H are the spatial dimensions.

We can feed a geometric tensor to an equivariant module as we feed normal tensors in *PyTorch*'s modules:

[65]: `y = conv(x)`

We can verify that the output is indeed associated with the output type of the convolutional layer:

[66]: `assert y.type == feat_type_out`

Lets check whether the output transforms as described by the output type when the input transforms according to the input type. The G -transformation of a geometric tensor is hereby conveniently done by calling `nn.GeometricTensor.transform()`.

```
[67]: # for each group element
for g in G.elements:
    # transform the input with the current group element according to the input type
    x_transformed = x.transform(g)

    # feed the transformed input in the convolutional layer
    y_from_x_transformed = conv(x_transformed)

    # the result should be equivalent to rotating the output produced in the
    # previous block according to the output type
    y_transformed_from_x = y.transform(g)
    assert torch.allclose(y_from_x_transformed.tensor, y_transformed_from_x.tensor, atol=1e-5), g
```

Any network operation is required to be equivariant. `escnn.nn` provides a wide range of equivariant network modules which guarantee this behavior.

Non-Linearities: As an example, we will next apply an *equivariant nonlinearity* to the output feature field of the convolution. Since the regular representations of a finite group G consists of permutation matrices, any pointwise nonlinearity like *ReLUs* is equivariant. Note that this is *not* the case for many other choices of representations / field types!

We instantiate a `escnn.nn.ReLU`, which, as an `nn.EquivariantModule`, requires to be informed about its input type to be able to perform the type checking. Here we are passing `feat_type_out`, the output of the equivariant convolution layer, as input type. It is not necessary to pass an output type to the nonlinearity since this is here determined by its input type.

[68]: `relu = nn.ReLU(feat_type_out)`

`z = relu(y)`

We can verify the equivariance again:

```
[69]: # for each group element
for g in G.elements:
    y_transformed = y.transform(g)
    z_from_y_transformed = relu(y_transformed)

    z_transformed_from_y = z.transform(g)

    assert torch.allclose(z_from_y_transformed.tensor, z_transformed_from_y.tensor, atol=1e-5), g
```

Deeper Models: In *deep learning* we usually want to stack multiple layers to build a deep model. As long as each layer is equivariant and consecutive layers are compatible, the equivariance property is preserved by induction.

The compatibility of two consecutive layers requires the output type of the first layer to be equal to the input type of the second layer.

In case we feed an input with the wrong type to a module, an error is raised:

```
[70]: layer1 = nn.R2Conv(feat_type_in, feat_type_out, kernel_size=3)
layer2 = nn.ReLU(feat_type_in) # the input type of the ReLU should be the output type of,
# the convolution

x = feat_type_in(torch.randn(3, 1, 7, 7))

try:
    y = layer2(layer1(x))
except AssertionError as e:
    print(e)

Error! the type of the input does not match the input type of this module
```

Simple deeper architectures can be built using a **SequentialModule**:

```
[71]: feat_type_in = nn.FieldType(r2_act, [r2_act.trivial_repr])
feat_type_hid = nn.FieldType(r2_act, 8*[r2_act.regular_repr])
feat_type_out = nn.FieldType(r2_act, 2*[r2_act.regular_repr])

model = nn.SequentialModule(
    nn.R2Conv(feat_type_in, feat_type_hid, kernel_size=3),
    nn.InnerBatchNorm(feat_type_hid),
    nn.ReLU(feat_type_hid, inplace=True),
    nn.R2Conv(feat_type_hid, feat_type_hid, kernel_size=3),
    nn.InnerBatchNorm(feat_type_hid),
    nn.ReLU(feat_type_hid, inplace=True),
    nn.R2Conv(feat_type_hid, feat_type_out, kernel_size=3),
).eval()
```

As every layer is equivariant and consecutive layers are compatible, the whole model is equivariant.

```
[72]: x = torch.randn(1, 1, 17, 17)
x = feat_type_in(x)

y = model(x)

# for each group element
for g in G.elements:
    x_transformed = x.transform(g)
    y_from_x_transformed = model(x_transformed)

    y_transformed_from_x = y.transform(g)

    assert torch.allclose(y_from_x_transformed.tensor, y_transformed_from_x.tensor, atol=1e-5), g
```

Invariant Pooling Layer: Usually, at the end of the model we want to produce a single feature vector to use for classification. To do so, it is common to pool over the spatial dimensions, e.g. via average pooling.

This produces (approximatively) translation-invariant feature vectors.

```
[73]: # average pooling with window size 11
avgpool = nn.PointwiseAvgPool(feat_type_out, 11)

y = avgpool(model(x))

print(y.shape)

torch.Size([1, 8, 1, 1])
```

In our case, the feature vectors $f(x) \in \mathbb{R}^c$ associated to each point $x \in \mathbb{R}^2$ have a well defined transformation law. The output of the model now transforms according to `feat_type_out` (here two C_4 regular fields, i.e. 8 channels). For our choice of regular representations (which are permutation representations) the channels in the feature vectors associated to each point permute when the input is rotated.

```
[74]: for g in G.elements:
    print(f'rotation by {g}:', y.transform(g).tensor[0, ...].detach().numpy().squeeze())

rotation by 0[2pi/4]: [-0.15 -0.172 -0.193 -0.185 -0.616 -0.632 -0.552 -0.597]
rotation by 1[2pi/4]: [-0.185 -0.15 -0.172 -0.193 -0.597 -0.616 -0.632 -0.552]
rotation by 2[2pi/4]: [-0.193 -0.185 -0.15 -0.172 -0.552 -0.597 -0.616 -0.632]
rotation by 3[2pi/4]: [-0.172 -0.193 -0.185 -0.15 -0.632 -0.552 -0.597 -0.616]
```

Many learning tasks require to build models which are **invariant** under rotations. We can compute invariant features from the output of the model using an **invariant map**. For instance, we can take the maximum value within each regular field. We do so using `nn.GroupPooling`:

```
[75]: invariant_map = nn.GroupPooling(feat_type_out)

y = invariant_map(avgpool(model(x)))

for g in G.elements:
    print(f'rotation by {g}:', y.transform(g).tensor[0, ...].detach().numpy().squeeze())

rotation by 0[2pi/4]: [-0.15 -0.552]
rotation by 1[2pi/4]: [-0.15 -0.552]
rotation by 2[2pi/4]: [-0.15 -0.552]
rotation by 3[2pi/4]: [-0.15 -0.552]
```

```
[76]: # for each group element
for g in G.elements:
    # rotated the input image
    x_transformed = x.transform(g)
    y_from_x_transformed = invariant_map(avgpool(model(x_transformed)))

    y_transformed_from_x = y # no .transform(g) needed since y should be invariant!

    # check that the output did not change
    # note that here we are not rotating the original output y as before
    assert torch.allclose(y_from_x_transformed.tensor, y_transformed_from_x.tensor,
                           atol=1e-6), g
```

2.3 Steerable CNN with infinite group G

We can now repeat the same constructions with G being an infinite group, e.g. the group of all planar rotations $G = SO(2)$.

```
[77]: # use N=-1 to indicate all rotations
```

```
r2_act = gspaces.rot2dOnR2(N=-1)
r2_act
```

```
[77]: SO(2)_on_R2[(None, -1)]
```

```
[78]: G = r2_act.fibergroup
```

```
G
```

```
[78]: SO(2)
```

```
[79]: # For simplicity we take a single-channel gray-scale image in input and we output a_
      ↳single-channel gray-scale image, i.e. we use scalar fields in input and output
feat_type_in = nn.FieldType(r2_act, [G.trivial_representation])
feat_type_out = nn.FieldType(r2_act, [G.trivial_representation])
```

As intermediate feature types, we want to use again the *regular representation*. Because G has an infinite number of elements, we use the Fourier transform idea described earlier. For example, we will use the first three irreps of $G = SO(2)$, which contains cosines and sines of frequency 0, 1 and 2. Earlier, we built this representation as

```
rho = G.spectral_regular_representation(*[G.irrep(f) for f in range(3)])
```

To apply a non-linearity, e.g. ELU, we can use the *Inverse Fourier Transform* to sample the function, apply the non-linearity and, finally, compute the *Fourier Transform* to recover the coefficients. Because G has infinite elements, the Fourier Transform requires an integral over G ; this can be **approximated** by a sum over a finite number of samples. The more samples one takes, the better the approximation will be, although this also increases the computational cost.

Fortunately, the class `nn.FourierELU` takes care of most of these details. We can just specify which irreps to consider, the number of channels (i.e. copies of the regular representation) and the number N of elements of G where to sample the function:

```
[80]: nonlinearity = nn.FourierELU(r2_act, 16, irreps=[(f,) for f in range(3)], N=12)
# we do not need to pre-define the feature type: FourierELU will create it internally_
# and we can just access it as
feat_type_hid = nonlinearity.in_type

# note also the its input and output types are the same
assert nonlinearity.in_type == nonlinearity.out_type
```

Let's build a simple $G = SO(2)$ equivariant model:

```
[81]: equivariant_so2_model = nn.SequentialModule(
    nn.R2Conv(feat_type_in, feat_type_hid, kernel_size=7),
    nn.IIDBatchNorm2d(feat_type_hid),
    nonlinearity,
    nn.R2Conv(feat_type_hid, feat_type_hid, kernel_size=7),
    nn.IIDBatchNorm2d(feat_type_hid),
    nonlinearity,
    nn.R2Conv(feat_type_hid, feat_type_out, kernel_size=7),
).eval()
```

and check its equivariance to a few elements of $SO(2)$:

```
[82]: x = torch.randn(1, 1, 23, 23)
x = feat_type_in(x)

y = equivariant_so2_model(x)

# check equivariance to N=16 rotations
N = 16

try:
    for i in range(N):
        g = G.element(i*2*np.pi/N)
        x_transformed = x.transform(g)
        y_from_x_transformed = equivariant_so2_model(x_transformed)

        y_transformed_from_x = y.transform(g)

        assert torch.allclose(y_from_x_transformed.tensor, y_transformed_from_x.tensor, atol=1e-3), g
except:
    print('Error! The model is not equivariant!')

```

Error! The model is not equivariant!

QUESTION 10

The model is not perfectly equivariant to $G = SO(2)$! Why is this an expected behaviour?

ANSWER 10

The $SO(2)$ group includes all continuous planar rotations. However, when an image is represented on a pixel grid, only the 4 rotations by angles multiple of $\pi/2$ are perfect, while other rotations involve some form of interpolation and generally introduce some noise. This prevents perfect equivariance to all rotations, since rotated versions of the same image inherently include some noise. A similar argument applies to the filters used during convolution: the steerability constraint described before involve a rotation of the filter k itself, but also the filter needs to be represented on discrete grid.

While the model can not be perfectly equivariant, we can compare it with a *conventional CNN* baseline. Let's build a CNN similar to our equivariant model but which is not constrained to be equivariant:

```
[83]: conventional_model = torch.nn.Sequential(
    torch.nn.Conv2d(feat_type_in.size, feat_type_hid.size, kernel_size=7),
    torch.nn.BatchNorm2d(feat_type_hid.size),
    torch.nn.ELU(),
    torch.nn.Conv2d(feat_type_hid.size, feat_type_hid.size, kernel_size=7),
    torch.nn.BatchNorm2d(feat_type_hid.size),
    torch.nn.ELU(),
```

(continues on next page)

(continued from previous page)

```
torch.nn.Conv2d(feat_type_hid.size, feat_type_out.size, kernel_size=7),
).eval()
```

To compare the two models, we compute their *equivariance error* for a few elements of G . We define the equivariance error of a model F with respect to a group element $g \in G$ and an input x as:

$$\epsilon_g(F) = \frac{\|F(g.X) - g.F(X)\|_2}{\|F(x)\|_2}$$

Note that this is a form of *relative* error. Let's now compute the equivariance error of the two models:

```
[84]: # let's generate a random image of shape W x W
W = 37
x = torch.randn(1, 1, W, W)

# Because a rotation by an angle smaller than 90 degrees moves pixels outsize the image, ↴
# we mask out all pixels outside the central disk
# We need to do this both for the input and the output

def build_mask(W):
    center_mask = np.zeros((2, W, W))
    center_mask[1, :, :] = np.arange(0, W) - W // 2
    center_mask[0, :, :] = np.arange(0, W) - W // 2
    center_mask[0, :, :] = center_mask[0, :, :].T
    center_mask = center_mask[0, :, :] ** 2 + center_mask[1, :, :] ** 2 < .9*(W // 2) ** 2
    center_mask = torch.tensor(center_mask.reshape(1, 1, W, W), dtype=torch.float)
    return center_mask

# create the mask for the input
input_center_mask = build_mask(W)

# mask the input image
x = x * input_center_mask
x = feat_type_in(x)

# compute the output of both models
y_equivariant = equivariant_so2_model(x)
y_conventional = feat_type_out(conventional_model(x.tensor))

# create the mask for the output images
output_center_mask = build_mask(y_equivariant.shape[-1])

# We evaluate the equivariance error on N=100 rotations
N = 100

error_equivariant = []
error_conventional = []

# for each of the N rotations
for i in range(N+1):
```

(continues on next page)

(continued from previous page)

```

g = G.element(i / N * 2*np.pi)

# rotate the input
x_transformed = x.transform(g)
x_transformed.tensor *= input_center_mask

# F(g.X) feed the transformed images in both models
y_from_x_transformed_equivariant = equivariant_so2_model(x_transformed).tensor
y_from_x_transformed_conventional = conventional_model(x_transformed.tensor)

# g.F(x) transform the output of both models
y_transformed_from_x_equivariant = y_equivariant.transform(g)
y_transformed_from_x_conventional = y_conventional.transform(g)

# mask all the outputs
y_from_x_transformed_equivariant = y_from_x_transformed_equivariant * output_center_
mask
y_from_x_transformed_conventional = y_from_x_transformed_conventional * output_
center_mask
y_transformed_from_x_equivariant = y_transformed_from_x_equivariant.tensor * output_
center_mask
y_transformed_from_x_conventional = y_transformed_from_x_conventional.tensor *_
output_center_mask

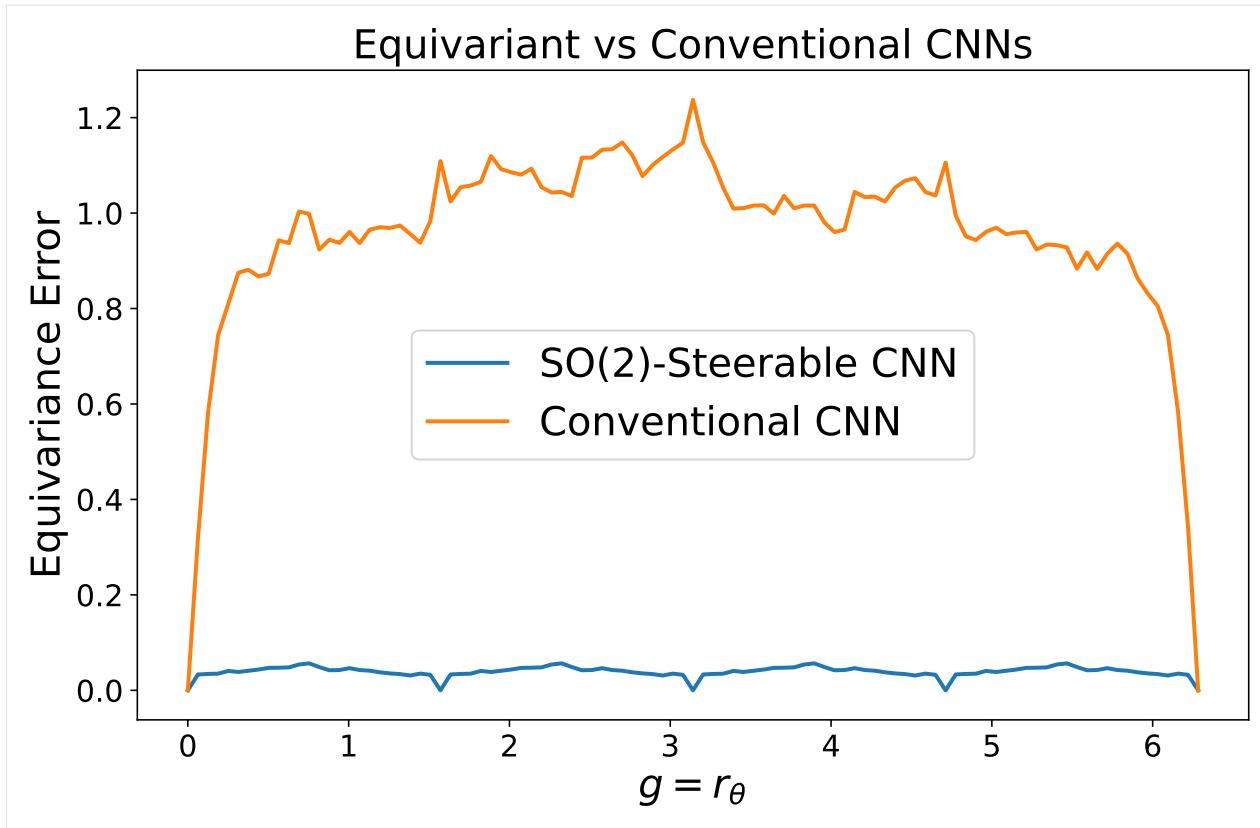
# compute the relative error of both models
rel_error_equivariant = torch.norm(y_from_x_transformed_equivariant - y_transformed_
from_x_equivariant).item() / torch.norm(y_equivariant.tensor).item()
rel_error_conventional = torch.norm(y_from_x_transformed_conventional - y_
transformed_from_x_conventional).item() / torch.norm(y_conventional.tensor).item()

error_equivariant.append(rel_error_equivariant)
error_conventional.append(rel_error_conventional)

# plot the error of both models as a function of the rotation angle theta
fig, ax = plt.subplots(figsize=(10, 6))

xs = [i*2*np.pi / N for i in range(N+1)]
plt.plot(xs, error_equivariant, label='SO(2)-Steerable CNN')
plt.plot(xs, error_conventional, label='Conventional CNN')
plt.title('Equivariant vs Conventional CNNs', fontsize=20)
plt.xlabel(r'$g = r\backslash\theta$', fontsize=20)
plt.ylabel('Equivariance Error', fontsize=20)
ax.tick_params(axis='both', which='major', labelsize=15)
plt.legend(fontsize=20)
plt.show()

```



4.31.4 3. Build and Train Steerable CNNs

Finally, we will proceed with implementing a **Steerable CNN** and train it on rotated MNIST.

Dataset

We will evaluate the model on the *rotated* MNIST dataset. First, we download the (non-rotated) MNIST 12k data:

```
[85]: # download the dataset
!wget -nc http://www.iro.umontreal.ca/~lisa/icml2007data/mnist.zip
# uncompress the zip file
!unzip -n mnist.zip -d mnist

File 'mnist.zip' already there; not retrieving.

Archive: mnist.zip
```

Then, we build the dataset and some utility functions:

```
[86]: from torch.utils.data import Dataset
from torchvision.transforms import RandomRotation
from torchvision.transforms import Pad
from torchvision.transforms import Resize
from torchvision.transforms import ToTensor
from torchvision.transforms import Compose
```

(continues on next page)

(continued from previous page)

```
from tqdm.auto import tqdm

from PIL import Image

device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

```
[87]: class MnistDataset(Dataset):

    def __init__(self, mode, rotated: bool = True):
        assert mode in ['train', 'test']

        if mode == "train":
            file = "mnist/mnist_train.amat"
        else:
            file = "mnist/mnist_test.amat"

        data = np.loadtxt(file)

        images = data[:, :-1].reshape(-1, 28, 28).astype(np.float32)

        # images are padded to have shape 29x29.
        # this allows to use odd-size filters with stride 2 when downsampling a feature ↵
        ↵map in the model
        pad = Pad((0, 0, 1, 1), fill=0)

        # to reduce interpolation artifacts (e.g. when testing the model on rotated ↵
        ↵images),
        # we upsample an image by a factor of 3, rotate it and finally downsample it ↵
        ↵again
        resize1 = Resize(87) # to upsample
        resize2 = Resize(29) # to downsample

        totensor = ToTensor()

        if rotated:
            self.images = torch.empty((images.shape[0], 1, 29, 29))
            for i in tqdm(range(images.shape[0]), leave=False):
                img = images[i]
                img = Image.fromarray(img, mode='F')
                r = (np.random.rand() * 360.)
                self.images[i] = totensor(resize2(resize1(pad(img))).rotate(r, Image.BILINEAR)).reshape(1, 29, 29)
        else:
            self.images = torch.zeros((images.shape[0], 1, 29, 29))
            self.images[:, :, :28, :28] = torch.tensor(images).reshape(-1, 1, 28, 28)

        self.labels = data[:, -1].astype(np.int64)
        self.num_samples = len(self.labels)

    def __getitem__(self, index):
        image, label = self.images[index], self.labels[index]
```

(continues on next page)

(continued from previous page)

```

    return image, label

def __len__(self):
    return len(self.labels)

```

```
[88]: # Set the random seed for reproducibility
np.random.seed(42)

# build the rotated training and test datasets
mnist_train = MnistDataset(mode='train', rotated=True)
train_loader = torch.utils.data.DataLoader(mnist_train, batch_size=64)

mnist_test = MnistDataset(mode='test', rotated=True)
test_loader = torch.utils.data.DataLoader(mnist_test, batch_size=64)

# for testing purpose, we also build a version of the test set with *non*-rotated digits
raw_mnist_test = MnistDataset(mode='test', rotated=False)
```

SO(2) equivariant architecture

We now build an *SO(2)* equivariant CNN.

Because the inputs are still gray-scale images, the input type of the model is again a *scalar field*. In the intermediate layers, we will use *regular fields*, such that the models are equivalent to *group-equivariant convolutional neural networks* (GCNNs).

The final classification is performed by a fully connected layer.

```
[89]: class S02SteerableCNN(torch.nn.Module):

    def __init__(self, n_classes=10):

        super(S02SteerableCNN, self).__init__()

        # the model is equivariant under all planar rotations
        self.r2_act = gspaces.rot2dOnR2(N=-1)

        # the input image is a scalar field, corresponding to the trivial representation
        in_type = nn.FieldType(self.r2_act, [self.r2_act.trivial_repr])

        # we store the input type for wrapping the images into a geometric tensor during
        # the forward pass
        self.input_type = in_type

        # We need to mask the input image since the corners are moved outside the grid
        # under rotations
        self.mask = nn.MaskModule(in_type, 29, margin=1)

        # convolution 1
        # first we build the non-linear layer, which also constructs the right feature
        # type
```

(continues on next page)

(continued from previous page)

```

# we choose 8 feature fields, each transforming under the regular representation
# of SO(2) up to frequency 3
# When taking the ELU non-linearity, we sample the feature fields on N=16 points
activation1 = nn.FourierELU(self.r2_act, 8, irreps=[(f,) for f in range(4)], N=16, inplace=True)
out_type = activation1.in_type
self.block1 = nn.SequentialModule(
    nn.R2Conv(in_type, out_type, kernel_size=7, padding=1, bias=False),
    nn.IIDBatchNorm2d(out_type),
    activation1,
)

# convolution 2
# the old output type is the input type to the next layer
in_type = self.block1.out_type
# the output type of the second convolution layer are 16 regular feature fields
activation2 = nn.FourierELU(self.r2_act, 16, irreps=[(f,) for f in range(4)], N=16, inplace=True)
out_type = activation2.in_type
self.block2 = nn.SequentialModule(
    nn.R2Conv(in_type, out_type, kernel_size=5, padding=2, bias=False),
    nn.IIDBatchNorm2d(out_type),
    activation2
)
# to reduce the downsampling artifacts, we use a Gaussian smoothing filter
self.pool1 = nn.SequentialModule(
    nn.PointwiseAvgPoolAntialiased(out_type, sigma=0.66, stride=2)
)

# convolution 3
# the old output type is the input type to the next layer
in_type = self.block2.out_type
# the output type of the third convolution layer are 32 regular feature fields
activation3 = nn.FourierELU(self.r2_act, 32, irreps=[(f,) for f in range(4)], N=16, inplace=True)
out_type = activation3.in_type
self.block3 = nn.SequentialModule(
    nn.R2Conv(in_type, out_type, kernel_size=5, padding=2, bias=False),
    nn.IIDBatchNorm2d(out_type),
    activation3
)

# convolution 4
# the old output type is the input type to the next layer
in_type = self.block3.out_type
# the output type of the fourth convolution layer are 64 regular feature fields
activation4 = nn.FourierELU(self.r2_act, 32, irreps=[(f,) for f in range(4)], N=16, inplace=True)
out_type = activation4.in_type
self.block4 = nn.SequentialModule(
    nn.R2Conv(in_type, out_type, kernel_size=5, padding=2, bias=False),
    nn.IIDBatchNorm2d(out_type),
)

```

(continues on next page)

(continued from previous page)

```

        activation4
    )
    self.pool2 = nn.SequentialModule(
        nn.PointwiseAvgPoolAntialiased(out_type, sigma=0.66, stride=2)
    )

    # convolution 5
    # the old output type is the input type to the next layer
    in_type = self.block4.out_type
    # the output type of the fifth convolution layer are 96 regular feature fields
    activation5 = nn.FourierELU(self.r2_act, 64, irreps=[(f,) for f in range(4)],  

→N=16, inplace=True)
    out_type = activation5.in_type
    self.block5 = nn.SequentialModule(
        nn.R2Conv(in_type, out_type, kernel_size=5, padding=2, bias=False),
        nn.IIDBatchNorm2d(out_type),
        activation5
    )

    # convolution 6
    # the old output type is the input type to the next layer
    in_type = self.block5.out_type
    # the output type of the sixth convolution layer are 64 regular feature fields
    activation6 = nn.FourierELU(self.r2_act, 64, irreps=[(f,) for f in range(4)],  

→N=16, inplace=True)
    out_type = activation6.in_type
    self.block6 = nn.SequentialModule(
        nn.R2Conv(in_type, out_type, kernel_size=5, padding=1, bias=False),
        nn.IIDBatchNorm2d(out_type),
        activation6
    )
    self.pool3 = nn.PointwiseAvgPoolAntialiased(out_type, sigma=0.66, stride=1,  

→padding=0)

    # number of output invariant channels
    c = 64

    # last 1x1 convolution layer, which maps the regular fields to c=64 invariant  

→scalar fields
    # this is essential to provide *invariant* features in the final classification  

→layer
    output_invariant_type = nn.FieldType(self.r2_act, c*[self.r2_act.trivial_repr])
    self.invariant_map = nn.R2Conv(out_type, output_invariant_type, kernel_size=1,  

→bias=False)

    # Fully Connected classifier
    self.fully_net = torch.nn.Sequential(
        torch.nn.BatchNorm1d(c),
        torch.nn.ELU(inplace=True),
        torch.nn.Linear(c, n_classes),
    )

```

(continues on next page)

(continued from previous page)

```

def forward(self, input: torch.Tensor):
    # wrap the input tensor in a GeometricTensor
    # (associate it with the input type)
    x = self.input_type(input)

    # mask out the corners of the input image
    x = self.mask(x)

    # apply each equivariant block

    # Each layer has an input and an output type
    # A layer takes a GeometricTensor in input.
    # This tensor needs to be associated with the same representation of the layer's
↳input type
    #
    # Each layer outputs a new GeometricTensor, associated with the layer's output
↳type.
    # As a result, consecutive layers need to have matching input/output types
    x = self.block1(x)
    x = self.block2(x)
    x = self.pool1(x)

    x = self.block3(x)
    x = self.block4(x)
    x = self.pool2(x)

    x = self.block5(x)
    x = self.block6(x)

    # pool over the spatial dimensions
    x = self.pool3(x)

    # extract invariant features
    x = self.invariant_map(x)

    # unwrap the output GeometricTensor
    # (take the Pytorch tensor and discard the associated representation)
    x = x.tensor

    # classify with the final fully connected layer
    x = self.fully_net(x.reshape(x.shape[0], -1))

return x

```

Equivariance Test before training

Let's instantiate the model:

```
[90]: model = S02SteerableCNN().to(device)
```

The model is now randomly initialized. Therefore, we do not expect it to produce the right class probabilities.

However, the model should still produce the same output for rotated versions of the same image. This is true for rotations by multiples of $\frac{\pi}{2}$, but is only approximate for other rotations.

Let's test it on a random test image: we feed $N = 20$ rotated versions of the first image in the test set and print the output logits of the model for each of them.

```
[91]: def test_model_single_image(model: torch.nn.Module, x: torch.Tensor, N: int = 8):
    np.set_printoptions(linewidth=10000)

    x = Image.fromarray(x.cpu().numpy()[0], mode='F')

    # to reduce interpolation artifacts (e.g. when testing the model on rotated images),
    # we upsample an image by a factor of 3, rotate it and finally downsample it again
    resize1 = Resize(87) # to upsample
    resize2 = Resize(29) # to downsample

    totensor = ToTensor()

    x = resize1(x)

    # evaluate the `model` on N rotated versions of the input image `x`
    model.eval()

    print()
    print('#####')
    header = 'angle | ' + ' '.join(['{:5d}'.format(d) for d in range(10)])
    print(header)
    with torch.no_grad():
        for r in range(N):
            x_transformed = totensor(resize2(x.rotate(r*360./N, Image.BILINEAR))).\
                reshape(1, 1, 29, 29)
            x_transformed = x_transformed.to(device)

            y = model(x_transformed)
            y = y.to('cpu').numpy().squeeze()

            angle = r * 360. / N
            print('{:6.1f} : {}'.format(angle, y))
    print('#####')
    print()
```

```
[92]: # retrieve the first image from the test set
x, y = next(iter(raw_mnist_test))

# evaluate the model
test_model_single_image(model, x, N=20)

#####
#  

angle | 0 1 2 3 4 5 6 7 8 9  

0.0 : [-0.359 1.492 0.49 -1.464 0.884 -1.195 -1.714 -0.802 2.39 -0.307]  

18.0 : [-0.371 1.505 0.513 -1.476 0.866 -1.205 -1.714 -0.807 2.387 -0.302]  

36.0 : [-0.364 1.518 0.511 -1.488 0.875 -1.205 -1.719 -0.817 2.415 -0.314]  

54.0 : [-0.347 1.524 0.5 -1.476 0.87 -1.196 -1.743 -0.803 2.409 -0.325]  

72.0 : [-0.347 1.496 0.476 -1.47 0.884 -1.198 -1.744 -0.791 2.385 -0.327]  

90.0 : [-0.359 1.492 0.49 -1.464 0.884 -1.195 -1.714 -0.802 2.39 -0.307]  

108.0 : [-0.371 1.505 0.513 -1.476 0.866 -1.205 -1.714 -0.807 2.387 -0.302]  

126.0 : [-0.364 1.518 0.511 -1.488 0.875 -1.205 -1.719 -0.817 2.415 -0.314]  

144.0 : [-0.347 1.524 0.5 -1.476 0.87 -1.196 -1.743 -0.803 2.409 -0.325]  

162.0 : [-0.347 1.496 0.476 -1.47 0.884 -1.198 -1.744 -0.791 2.385 -0.327]  

180.0 : [-0.359 1.492 0.49 -1.464 0.884 -1.195 -1.714 -0.802 2.39 -0.307]  

198.0 : [-0.371 1.505 0.513 -1.476 0.866 -1.205 -1.714 -0.807 2.387 -0.302]  

216.0 : [-0.364 1.518 0.511 -1.488 0.875 -1.205 -1.719 -0.817 2.415 -0.314]  

234.0 : [-0.347 1.524 0.5 -1.476 0.87 -1.196 -1.743 -0.803 2.409 -0.325]  

252.0 : [-0.347 1.496 0.476 -1.47 0.884 -1.198 -1.744 -0.791 2.385 -0.327]  

270.0 : [-0.359 1.492 0.49 -1.464 0.884 -1.195 -1.714 -0.802 2.39 -0.307]  

288.0 : [-0.371 1.505 0.513 -1.476 0.866 -1.205 -1.714 -0.807 2.387 -0.302]  

306.0 : [-0.364 1.518 0.511 -1.488 0.875 -1.205 -1.719 -0.817 2.415 -0.314]  

324.0 : [-0.347 1.524 0.5 -1.476 0.87 -1.196 -1.743 -0.803 2.409 -0.325]  

342.0 : [-0.347 1.496 0.476 -1.47 0.884 -1.198 -1.744 -0.791 2.385 -0.327]
#####
#
```

The output of the model is already almost invariant but we observe small fluctuations in the outputs. This is the effect of the discretization artifacts (e.g. the pixel grid can not be perfectly rotated by any angle without interpolation) and can not be completely removed.

Training the model

Let's train the model now. The procedure is the same used to train a normal *PyTorch* architecture:

```
[93]: # build the training and test function

def test(model: torch.nn.Module):
    # test over the full rotated test set
    total = 0
    correct = 0

    with torch.no_grad():
        model.eval()
        for i, (x, t) in enumerate(test_loader):
```

(continues on next page)

(continued from previous page)

```

x = x.to(device)
t = t.to(device)

y = model(x)

_, prediction = torch.max(y.data, 1)
total += t.shape[0]
correct += (prediction == t).sum().item()
return correct/total*100.

def train(model: torch.nn.Module, lr=1e-4, wd=1e-4, checkpoint_path: str = None):
    if checkpoint_path is not None:
        checkpoint_path = os.path.join(CHECKPOINT_PATH, checkpoint_path)

    if checkpoint_path is not None and os.path.isfile(checkpoint_path):
        model.load_state_dict(torch.load(checkpoint_path))
        model.eval()
        return

    loss_function = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=wd)

    for epoch in tqdm(range(21)):
        model.train()
        for i, (x, t) in enumerate(train_loader):
            optimizer.zero_grad()

            x = x.to(device)
            t = t.to(device)

            y = model(x)

            loss = loss_function(y, t)

            loss.backward()

            optimizer.step()
            del x, y, t, loss

        if epoch % 10 == 0:
            accuracy = test(model)
            print(f"epoch {epoch} | test accuracy: {accuracy}")

    if checkpoint_path is not None:
        torch.save(model.state_dict(), checkpoint_path)

```

Finally, train the $SO(2)$ equivariant model:

```
[94]: # set the seed manually for reproducibility
torch.manual_seed(42)
model = SO2SteerableCNN().to(device)
```

(continues on next page)

(continued from previous page)

```
train(model, checkpoint_path="steerable_so2-pretrained.ckpt")

accuracy = test(model)
print(f"Test accuracy: {accuracy}")

Test accuracy: 94.982
```

```
[95]: def test_model_rotations(model: torch.nn.Module, N: int = 24, M: int = 2000, checkpoint_
      ↵path: str = None):
    # evaluate the `model` on N rotated versions of the first M images in the test set

    if checkpoint_path is not None:
        checkpoint_path = os.path.join(CHECKPOINT_PATH, checkpoint_path)

    if checkpoint_path is not None and os.path.isfile(checkpoint_path):
        accuracies = np.load(checkpoint_path)
        return accuracies.tolist()

    model.eval()

    # to reduce interpolation artifacts (e.g. when testing the model on rotated images),
    # we upsample an image by a factor of 3, rotate it and finally downsample it again
    resize1 = Resize(87) # to upsample
    resize2 = Resize(29) # to downsample

    totensor = ToTensor()

    accuracies = []
    with torch.no_grad():
        model.eval()

        for r in tqdm(range(N)):
            total = 0
            correct = 0

            for i in range(M):
                x, t = raw_mnist_test[i]

                x = Image.fromarray(x.numpy()[0], mode='F')

                x = totensor(resize2(resize1(x).rotate(r*360./N, Image.BILINEAR))).\
                    ↵reshape(1, 1, 29, 29).to(device)

                x = x.to(device)

                y = model(x)

                _, prediction = torch.max(y.data, 1)
                total += 1
                correct += (prediction == t).sum().item()
```

(continues on next page)

(continued from previous page)

```

    accuracies.append(correct/total*100.)

    if checkpoint_path is not None:
        np.save(checkpoint_path, np.array(accuracies))

    return accuracies

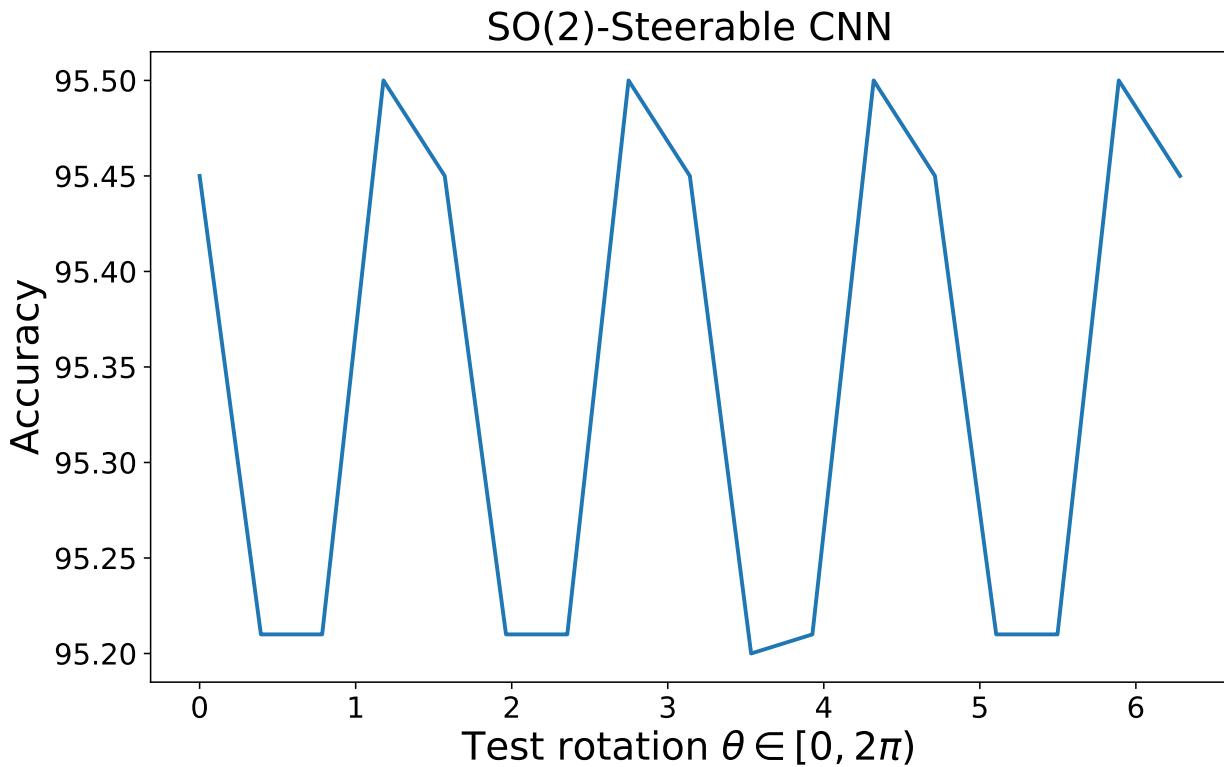
```

[96]: accs_so2 = test_model_rotations(model, 16, 10000, checkpoint_path="steerable_so2-
→ accuracies.npy")

[97]: # plot the accuracy of as a function of the rotation angle theta applied to the test set
fig, ax = plt.subplots(figsize=(10, 6))

N = 16

xs = [i*2*np.pi / N for i in range(N+1)]
plt.plot(xs, accs_so2 + [accs_so2[0]])
plt.title('SO(2)-Steerable CNN', fontsize=20)
plt.xlabel(r'Test rotation \$\theta \in [0, 2\pi]\$', fontsize=20)
plt.ylabel('Accuracy', fontsize=20)
ax.tick_params(axis='both', which='major', labelsize=15)
plt.show()



Even after training, the model is not perfectly $SO(2)$ equivariant, but we observe the accuracy is rather stable to rotations.

C_4 equivariant architecture

For comparison, let's build a similar architecture equivariant only to $N = 4$ rotations.

```
[98]: class CNSteerableCNN(torch.nn.Module):

    def __init__(self, n_classes=10):

        super(CNSteerableCNN, self).__init__()

        # the model is equivariant to rotations by multiples of  $2\pi/N$ 
        self.r2_act = gspaces.rot2dOnR2(N=4)

        # the input image is a scalar field, corresponding to the trivial representation
        in_type = nn.FieldType(self.r2_act, [self.r2_act.trivial_repr])

        # we store the input type for wrapping the images into a geometric tensor during
        # the forward pass
        self.input_type = in_type

        # We need to mask the input image since the corners are moved outside the grid
        # under rotations
        self.mask = nn.MaskModule(in_type, 29, margin=1)

        # convolution 1
        # first we build the non-linear layer, which also constructs the right feature
        # type
        # we choose 8 feature fields, each transforming under the regular representation
        # of  $C_4$ 
        activation1 = nn.ELU(nn.FieldType(self.r2_act, 8*[self.r2_act.regular_repr]),
        inplace=True)
        out_type = activation1.in_type
        self.block1 = nn.SequentialModule(
            nn.R2Conv(in_type, out_type, kernel_size=7, padding=1, bias=False),
            nn.IIDBatchNorm2d(out_type),
            activation1,
        )

        # convolution 2
        # the old output type is the input type to the next layer
        in_type = self.block1.out_type
        # the output type of the second convolution layer are 16 regular feature fields
        activation2 = nn.ELU(nn.FieldType(self.r2_act, 16*[self.r2_act.regular_repr]),
        inplace=True)
        out_type = activation2.in_type
        self.block2 = nn.SequentialModule(
            nn.R2Conv(in_type, out_type, kernel_size=5, padding=2, bias=False),
            nn.IIDBatchNorm2d(out_type),
            activation2
        )
        self.pool1 = nn.SequentialModule(
            nn.PointwiseAvgPoolAntialiased(out_type, sigma=0.66, stride=2)
        )
```

(continues on next page)

(continued from previous page)

```

# convolution 3
# the old output type is the input type to the next layer
in_type = self.block2.out_type
# the output type of the third convolution layer are 32 regular feature fields
activation3 = nn.ELU(nn.FieldType(self.r2_act, 32*[self.r2_act.regular_repr]),  

↪inplace=True)
out_type = activation3.in_type
self.block3 = nn.SequentialModule(
    nn.R2Conv(in_type, out_type, kernel_size=5, padding=2, bias=False),
    nn.IIDBatchNorm2d(out_type),
    activation3
)

# convolution 4
# the old output type is the input type to the next layer
in_type = self.block3.out_type
# the output type of the fourth convolution layer are 32 regular feature fields
activation4 = nn.ELU(nn.FieldType(self.r2_act, 32*[self.r2_act.regular_repr]),  

↪inplace=True)
out_type = activation4.in_type
self.block4 = nn.SequentialModule(
    nn.R2Conv(in_type, out_type, kernel_size=5, padding=2, bias=False),
    nn.IIDBatchNorm2d(out_type),
    activation4
)
self.pool2 = nn.SequentialModule(
    nn.PointwiseAvgPoolAntialiased(out_type, sigma=0.66, stride=2)
)

# convolution 5
# the old output type is the input type to the next layer
in_type = self.block4.out_type
# the output type of the fifth convolution layer are 64 regular feature fields
activation5 = nn.ELU(nn.FieldType(self.r2_act, 64*[self.r2_act.regular_repr]),  

↪inplace=True)
out_type = activation5.in_type
self.block5 = nn.SequentialModule(
    nn.R2Conv(in_type, out_type, kernel_size=5, padding=2, bias=False),
    nn.IIDBatchNorm2d(out_type),
    activation5
)

# convolution 6
# the old output type is the input type to the next layer
in_type = self.block5.out_type
# the output type of the sixth convolution layer are 64 regular feature fields
activation6 = nn.ELU(nn.FieldType(self.r2_act, 64*[self.r2_act.regular_repr]),  

↪inplace=True)
out_type = activation6.in_type
self.block6 = nn.SequentialModule(
    nn.R2Conv(in_type, out_type, kernel_size=5, padding=1, bias=False),

```

(continues on next page)

(continued from previous page)

```

        nn.IIDBatchNorm2d(out_type),
        activation6
    )
    self.pool3 = nn.PointwiseAvgPoolAntialiased(out_type, sigma=0.66, stride=1,
→padding=0)

    # number of output invariant channels
    c = 64

    output_invariant_type = nn.FieldType(self.r2_act, c*[self.r2_act.trivial_repr])
    self.invariant_map = nn.R2Conv(out_type, output_invariant_type, kernel_size=1,
→bias=False)

    # Fully Connected classifier
    self.fully_net = torch.nn.Sequential(
        torch.nn.BatchNorm1d(c),
        torch.nn.ELU(inplace=True),
        torch.nn.Linear(c, n_classes),
    )

def forward(self, input: torch.Tensor):
    # wrap the input tensor in a GeometricTensor
    # (associate it with the input type)
    x = self.input_type(input)

    # mask out the corners of the input image
    x = self.mask(x)

    # apply each equivariant block

    # Each layer has an input and an output type
    # A layer takes a GeometricTensor in input.
    # This tensor needs to be associated with the same representation of the layer's
→input type
    #
    # Each layer outputs a new GeometricTensor, associated with the layer's output
→type.
    # As a result, consecutive layers need to have matching input/output types
    x = self.block1(x)
    x = self.block2(x)
    x = self.pool1(x)

    x = self.block3(x)
    x = self.block4(x)
    x = self.pool2(x)

    x = self.block5(x)
    x = self.block6(x)

    # pool over the spatial dimensions
    x = self.pool3(x)

```

(continues on next page)

(continued from previous page)

```

# extract invariant features
x = self.invariant_map(x)

# unwrap the output GeometricTensor
# (take the Pytorch tensor and discard the associated representation)
x = x.tensor

# classify with the final fully connected layer
x = self.fully_net(x.reshape(x.shape[0], -1))

return x

```

Instantiate and train the C_4 equivariant model:

```
[99]: torch.manual_seed(42)
model_c4 = CNSteerableCNN().to(device)
train(model_c4, checkpoint_path="steerable_c4-pretrained.ckpt")

accuracy = test(model_c4)
print(f"Test accuracy: {accuracy}")

accs_c4 = test_model_rotations(model_c4, 16, 10000, checkpoint_path="steerable_c4-
→accuracies.npy")

```

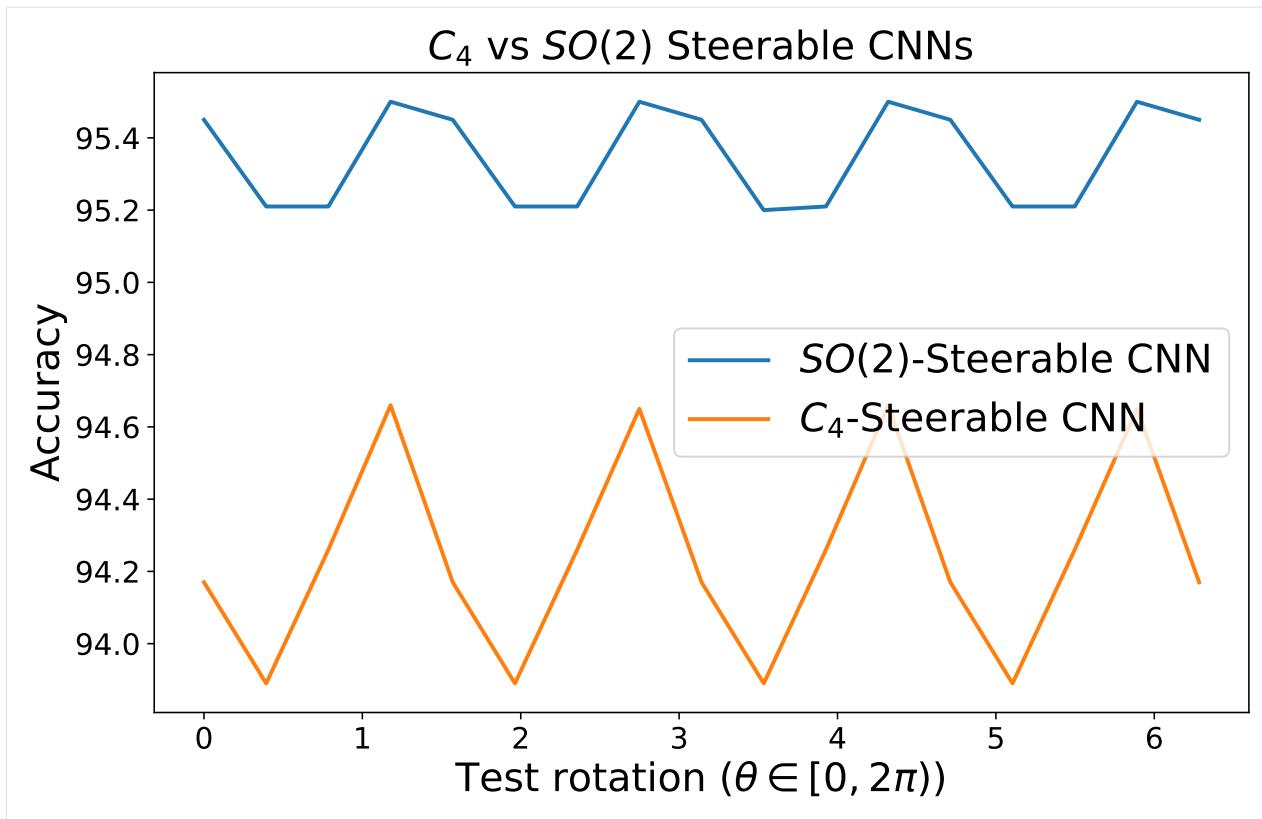
Test accuracy: 93.842

Finally, let's compare the performance of both models on the rotated test sets:

```
[100]: # plot the accuracy of as a function of the rotation angle theta applied to the test set
fig, ax = plt.subplots(figsize=(10, 6))

N=16

xs = [i*2*np.pi / N for i in range(N+1)]
plt.plot(xs, accs_so2 + [accs_so2[0]], label=r'$SO(2)$-Steerable CNN')
plt.plot(xs, accs_c4 + [accs_c4[0]], label=r'$C_4$-Steerable CNN')
plt.title(r'$C_4$ vs $SO(2)$ Steerable CNNs', fontsize=20)
plt.xlabel(r'Test rotation ($\theta \in [0, 2\pi]$)', fontsize=20)
plt.ylabel('Accuracy', fontsize=20)
ax.tick_params(axis='both', which='major', labelsize=15)
plt.legend(fontsize=20)
plt.show()
```



While perfect equivariance to $SO(2)$ is not achievable due to the discretizations, the $SO(2)$ equivariant architecture is more stable over the rotations of the test set than the C_4 model. Moreover, since C_4 is the only perfect symmetry of the pixel grid and since $C_4 < SO(2)$, the $SO(2)$ equivariant architecture is also perfectly equivariant to rotations by multiples of $\pi/2$.

4.31.5 Conclusion

In this tutorial, you first learnt about *group representation theory* and the *Fourier Transform* over compact groups. These are the mathematical tools used to formalize Steerable CNNs.

In the second part of this tutorial, you learnt about *steerable feature fields* and *steerable CNNs*. In particular, the previously defined Fourier transform allowed us to build a steerable CNN which is equivalent to a Group-Convolutional Neural Network (GCNN) equivariant to translations and the continuous group $G = SO(2)$ of rotations.

In our steerable CNNs, we mostly leveraged the *regular representation* of the group G , but the framework of steerable CNNs allows for a variety of representations. If you are interested in knowing more about steerable CNNs, this is a (non-exhaustive) list of relevant works you can check out:

- Steerable CNNs
- Harmonic Networks: Deep Translation and Rotation Equivariance
- 3D Steerable CNNs
- Tensor Field Networks
- A General Theory of Equivariant CNNs on Homogeneous Spaces
- Cormorant: Covariant Molecular Neural Networks
- General E(2)-Equivariant Steerable CNNs

- A Program to Build E(N)-Equivariant Steerable CNNs

4.32 DPM1 - Deep Probabilistic Models I

Filled notebook:

Recordings: Lecture 1.1 & Lecture 1.2

Authors: Wilker Aziz & Bryan Eikema

4.32.1 0. General notes

This notebook illustrates the concepts discussed in the module **Deep probabilistic models I** offered within DL2.

The examples in the notebook are based on NLP datasets, but the concepts are general enough that you can apply them in any domain. At the end of the notebook, we invite you to try your favourite dataset (as long as you can design an encoder for the data type you are interested in, transferring ideas from this notebook should be fairly simple).

The notebook starts with modelling univariate response variables conditionally, given a high-dimensional and structured input. We look into a nominal response variable and then into a numerical one. Next, we look into a structured response variable.

0.1 ILOs

- Prescribe joint distributions using PyTorch
- Estimate the parameters of the model via maximum likelihood estimation
- Implement a decision rule

0.2 Setting up

```
[1]: import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0

import random
import numpy as np
import torch

try:
    import nltk
    import sklearn
except ModuleNotFoundError: # Install nltk and sklearn if necessary
    !pip install --quiet nltk, sklearn
    import nltk

def seed_all(seed=42):
    np.random.seed(seed)
```

(continues on next page)

(continued from previous page)

```
random.seed(seed)
torch.manual_seed(seed)

seed_all()
```

4.32.2 1. Data

In this tutorial we will design conditional models involving structured data. We use text as an example and design

1. a classifier
2. an ordinal regressor
3. a sequence labeller

At the end of the tutorial you are encouraged to design similar models for a different type of data (for example, consider a different modality of data).

```
[2]: nltk.download('treebank')
nltk.download('brown')
nltk.download('punkt')
nltk.download('universal_tagset')

[nltk_data] Downloading package treebank to /home/phillip/nltk_data...
[nltk_data]   Package treebank is already up-to-date!
[nltk_data] Downloading package brown to /home/phillip/nltk_data...
[nltk_data]   Unzipping corpora/brown.zip.
[nltk_data] Downloading package punkt to /home/phillip/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package universal_tagset to
[nltk_data]   /home/phillip/nltk_data...
[nltk_data]   Package universal_tagset is already up-to-date!
```

```
[2]: True
```

1.1 Sentiment

```
[3]: from nltk.corpus import treebank, brown
```

The brown corpus contains texts organized into 15 genres.

```
[4]: brown.categories()
```

```
[4]: ['adventure',
 'belles_lettres',
 'editorial',
 'fiction',
 'government',
 'hobbies',
 'humor',
 'learned',
 'lore',
 'mystery',
```

(continues on next page)

(continued from previous page)

```
'news',
'religion',
'reviews',
'romance',
'science_fiction']
```

The dataset is already preprocessed for us:

```
[5]: print("fiction examples")
for i, x in zip(range(3), brown.sents(categories=['fiction'])):
    print(i, x)
print()
print("religion examples")
for i, x in zip(range(3), brown.sents(categories=['religion'])):
    print(i, x)
print()
print("learned examples")
for i, x in zip(range(3), brown.sents(categories=['learned'])):
    print(i, x)

'fiction' examples
0 ['Thirty-three']
1 ['Scotty', 'did', 'not', 'go', 'back', 'to', 'school', '.']
2 ['His', 'parents', 'talked', 'seriously', 'and', 'lengthily', 'to', 'their', 'own',
   ↪ 'doctor', 'and', 'to', 'a', 'specialist', 'at', 'the', 'University', 'Hospital', '--',
   ↪ 'Mr.', 'McKinley', 'was', 'entitled', 'to', 'a', 'discount', 'for', 'members', 'of',
   ↪ 'his', 'family', '--', 'and', 'it', 'was', 'decided', 'it', 'would', 'be', 'best', 'for',
   ↪ ', 'him', 'to', 'take', 'the', 'remainder', 'of', 'the', 'term', 'off', ',', 'spend',
   ↪ 'a', 'lot', 'of', 'time', 'in', 'bed', 'and', ',', 'for', 'the', 'rest', ',', 'do',
   ↪ 'pretty', 'much', 'as', 'he', 'chose', '--', 'provided', ',', 'of', 'course', ',', 'he
   ↪ ', 'chose', 'to', 'do', 'nothing', 'too', 'exciting', 'or', 'too', 'debilitating', '.']

'religion' examples
0 ['As', 'a', 'result', ',', 'although', 'we', 'still', 'make', 'use', 'of', 'this',
   ↪ 'distinction', ',', 'there', 'is', 'much', 'confusion', 'as', 'to', 'the', 'meaning',
   ↪ 'of', 'the', 'basic', 'terms', 'employed', '.']
1 ['Just', 'what', 'is', 'meant', 'by', '``', 'spirit', "''", 'and', 'by', '``', 'matter
   ↪ ', "''", '?', '?']
2 ['The', 'terms', 'are', 'generally', 'taken', 'for', 'granted', 'as', 'though', 'they',
   ↪ 'referred', 'to', 'direct', 'and', 'axiomatic', 'elements', 'in', 'the', 'common',
   ↪ 'experience', 'of', 'all', '.']

'learned' examples
0 ['1', '.']
1 ['Introduction']
2 ['It', 'has', 'recently', 'become', 'practical', 'to', 'use', 'the', 'radio', 'emission
   ↪ ', 'of', 'the', 'moon', 'and', 'planets', 'as', 'a', 'new', 'source', 'of',
   ↪ 'information', 'about', 'these', 'bodies', 'and', 'their', 'atmospheres', '.']

[6]: def split_nltk_categorised_corpus(corpus, categories, max_length=30, num_heldout=100):
    """
    Shuffle and split a corpus.

```

(continues on next page)

(continued from previous page)

```

corpus: a corpus of tagged sequences, each sequence is a pair, each pair is a token_
→ and a tag.
max_length: discard sentences longer than this

Return:
    (training word sequences, training tag sequences),
    (dev word sequences, dev tag sequences),
    (test word sequences, test tag sequences),
    """
sentences = []
labels = []
for k, c in enumerate(categories):
    seqs = corpus.sents(categories=[c])
    sentences.extend(seqs)
    labels.extend(len(seqs) * [k])
# do not change the seed in here
order = np.random.RandomState(42).permutation(np.arange(len(sentences)))
shuffled_sentences = [[w for w in sentences[i]] for i in order if len(sentences[i]) ≤ max_length]
shuffled_labels = [labels[i] for i in order if len(sentences[i]) ≤ max_length]
return (shuffled_sentences[2*num_heldout:], shuffled_labels[2*num_heldout:]),
→ (shuffled_sentences[num_heldout:2*num_heldout], shuffled_labels[num_heldout:2*num_
→ heldout]), (shuffled_sentences[:num_heldout], shuffled_labels[:num_heldout])

```

[7]:

```

%%time
(cat_training_x, cat_training_y), (cat_dev_x, cat_dev_y), (cat_test_x, cat_test_y) = split_nltk_categorised_corpus(brown, brown.categories(), num_heldout=1000)
print(f"Number of sentences: training={len(cat_training_x)} dev={len(cat_dev_x)} test={len(cat_test_x)}")

```

Number of sentences: training=44685 dev=1000 test=1000
CPU times: user 3.07 s, sys: 59.2 ms, total: 3.13 s
Wall time: 3.13 s

[8]:

```
len(cat_training_x), len(cat_dev_x), len(cat_test_x)
```

[8]:

```
(44685, 1000, 1000)
```

[9]:

```
cat_training_y[0], cat_training_x[0]
```

[9]:

```
(1,
['That',
 'is',
 'why',
 'the',
 'members',
 'of',
 'the',
 'beat',
 'generation',
 'proudly',
 'assume',
```

(continues on next page)

(continued from previous page)

```
'the',
'title',
'of',
'the',
'holy',
'barbarians',
';',
';'])
```

1.2 Age

Schler et al (2006) collected a dataset of blog posts annotated for the age of the author, we will use a subset of that dataset. The complete dataset can be found [on Kaggle](#).

Warning: we will use this dataset to illustrate Poisson regression, but note that in general age may be a protected attribute in certain applications, and one should carefully consider the implications of designing and deploying an age detection system.

```
[10]: # Download the data
!wget https://surfdrive.surf.nl/files/index.php/s/2xWdFxnewjN9gsq/download -O blog-
authorship.json.gz
!gzip -d blog-authorship.json.gz

--2022-04-05 17:42:50-- https://surfdrive.surf.nl/files/index.php/s/2xWdFxnewjN9gsq/
--download
Resolving surfdrive.surf.nl (surfdrive.surf.nl)... 145.100.27.67, 2001:610:108:203b:0:
-a11:da7a:5afe
Connecting to surfdrive.surf.nl (surfdrive.surf.nl)|145.100.27.67|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 25699440 (25M) [application/gzip]
Saving to: 'blog-authorship.json.gz'

blog-authorship.json 100%[=====] 24.51M 16.1MB/s in 1.5s

2022-04-05 17:42:52 (16.1 MB/s) - 'blog-authorship.json.gz' saved [25699440/25699440]
```

```
[11]: import json

with open("blog-authorship.json") as f:
    blog_data = json.load(f)
```

```
[12]: blog_data.keys()
[12]: dict_keys(['source', 'ack', 'subset', 'format', 'training', 'dev', 'test'])
```

```
[13]: blog_data['source'], blog_data['ack']
[13]: ('[Blog authorship corpus](https://www.kaggle.com/rstatman/blog-authorship-corpus)', '[Schler et al (2006)](http://www.cs.biu.ac.il/~schler_springsymp06.pdf)')
```

We already have a training/dev/test split, each data point is a pair and the text is not preprocessed for us:

```
[14]: len(blog_data['training']), len(blog_data['dev']), len(blog_data['test'])
```

```
[14]: (200000, 40000, 49174)
```

```
[15]: blog_data['training'][0]
```

```
[15]: ['I hate exams considering I just failed one. Yippee', 16]
```

We need to tokenize the data for use in deep learning. We will use the NLTK word tokenizer for this. We also lowercase the inputs and shuffle the training data. This will take some 3 minutes.

```
[16]: %time
from nltk.tokenize import word_tokenize

# Tokenize and shuffle the data.
order = np.random.RandomState(42).permutation(np.arange(len(blog_data["training"])))
blog_training_x = [word_tokenize(blog_data["training"][i][0].lower()) for i in order]
blog_training_y = [[blog_data["training"][i][1]] for i in order]
blog_dev_x = [word_tokenize(x[0].lower()) for x in blog_data["dev"]]
blog_dev_y = [[x[1]] for x in blog_data["dev"]]
blog_test_x = [word_tokenize(x[0].lower()) for x in blog_data["test"]]
blog_test_y = [[x[1]] for x in blog_data["test"]]

CPU times: user 1min 10s, sys: 306 ms, total: 1min 11s
Wall time: 1min 11s
```

```
[17]: # Example pre-processed data point
print(f"input: {blog_training_x[0]}")
print(f"age of author: {blog_training_y[0]}")

input: ['urllink', 'this', 'is', 'my', 'friend', 'sam', '.', 'he', "'s", 'a', 'little',
       'weird', '.', 'he', "'s", 'nice', 'though', '.', 'but', 'weird', '.']
age of author: [17]
```

1.3 Syntactic categories

From NLTK, we will also take the treebank corpus where words are annotated with their syntactic categories (parts of speech, or POS tags).

The method `tagged_sents` will give us a view of tokenized sentences with their token lag tag annotation:

```
[18]: example = treebank.tagged_sents(tagset='universal')[0] # 'universal' here refers to the_
        style of tags
example
```

```
[18]: [('Pierre', 'NOUN'),
       ('Vinken', 'NOUN'),
       ('.', '.'),
       ('61', 'NUM'),
       ('years', 'NOUN'),
       ('old', 'ADJ'),
       ('.', '.'),
       ('will', 'VERB'),
       ('join', 'VERB'),
```

(continues on next page)

(continued from previous page)

```
('the', 'DET'),
('board', 'NOUN'),
('as', 'ADP'),
('a', 'DET'),
('nonexecutive', 'ADJ'),
('director', 'NOUN'),
('Nov.', 'NOUN'),
('29', 'NUM'),
('. ', '.'))]
```

```
[19]: def split_nltk_tagged_corpus(corpus, max_length=30, num_heldout=100):
    """
    Shuffle and split a corpus.
    corpus: a corpus of tagged sequences, each sequence is a pair, each pair is a token
    ↪ and a tag.
    max_length: discard sentences longer than this

    Return:
        (training word sequences, training tag sequences),
        (dev word sequences, dev tag sequences),
        (test word sequences, test tag sequences),
    """
    tagged_sentences = corpus.tagged_sents(tagset='universal')
    # do not change the seed in here
    order = np.random.RandomState(42).permutation(np.arange(len(tagged_sentences)))
    word_sequences = [[w.lower() for w, t in tagged_sentences[i]] for i in order if
    ↪ len(tagged_sentences[i]) <= max_length]
    tag_sequences = [[t for w, t in tagged_sentences[i]] for i in order if len(tagged_
    ↪ sentences[i]) <= max_length]
    return (word_sequences[2*num_heldout:], tag_sequences[2*num_heldout:]), (word_
    ↪ sequences[num_heldout:2*num_heldout], tag_sequences[num_heldout:2*num_heldout]), (word_
    ↪ sequences[:num_heldout], tag_sequences[:num_heldout])
```

For treebank this will take about 10 seconds.

```
[20]: %time
(tagger_training_x, tagger_training_y), (tagger_dev_x, tagger_dev_y), (tagger_test_x,
    ↪ tagger_test_y) = split_nltk_tagged_corpus(treebank, num_heldout=100)
print(f"Number of sentences: training={len(tagger_training_x)} dev={len(tagger_dev_x)} ↪
    ↪ test={len(tagger_test_x)}")
```

Number of sentences: training=2486 dev=100 test=100
CPU times: user 2.38 s, sys: 116 ms, total: 2.5 s
Wall time: 2.52 s

```
[21]: print("# A few training sentences\n")
for n in range(3):
    print(f"x_{n} = {tagger_training_x[n]}")
    print(f"y_{n} = {tagger_training_y[n]}")
    print()

# A few training sentences
```

(continues on next page)

(continued from previous page)

```

x_0 = ['they', 'know', '0', 'he', 'is', 'generally', 'opposed', 'to', 'cop-killer',
       ↪'bullets', ',', 'but', 'that', 'he', 'had', 'some', 'reservations', 'about', 'the',
       ↪'language', 'in', 'the', 'legislation', '.', "''"]
y_0 = ['PRON', 'VERB', 'X', 'PRON', 'VERB', 'ADV', 'VERB', 'PRT', 'ADJ', 'NOUN', '.',
       ↪'CONJ', 'ADP', 'PRON', 'VERB', 'DET', 'NOUN', 'ADP', 'DET', 'NOUN', 'ADP', 'DET', 'NOUN
       ↪', '.', '.']

x_1 = ['california', "'s", 'education', 'department', 'suspects', 'adult',
       ↪'responsibility', 'for', 'erasures', 'at', '40', 'schools', 'that', '*t*-85', 'changed
       ↪', 'wrong', 'answers', 'to', 'right', 'ones', 'on', 'a', 'statewide', 'test', '.']
y_1 = ['NOUN', 'PRT', 'NOUN', 'NOUN', 'VERB', 'NOUN', 'NOUN', 'ADP', 'NOUN', 'ADP', 'NUM
       ↪', 'NOUN', 'DET', 'X', 'VERB', 'ADJ', 'NOUN', 'PRT', 'ADJ', 'NOUN', 'ADP', 'DET', 'ADJ
       ↪', 'NOUN', '.']

x_2 = ['the', 'loan', 'may', 'be', 'extended', '*-1', 'by', 'the', 'mcalpine', 'group',
       ↪'for', 'an', 'additional', 'year', 'with', 'an', 'increase', 'in', 'the', 'conversion',
       ↪'price', 'to', '$', '2.50', '*u*', 'a', 'share', '.']
y_2 = ['DET', 'NOUN', 'VERB', 'VERB', 'VERB', 'X', 'ADP', 'DET', 'NOUN', 'NOUN', 'ADP',
       ↪'DET', 'ADJ', 'NOUN', 'ADP', 'DET', 'NOUN', 'ADP', 'DET', 'NOUN', 'NOUN', 'PRT', '.',
       ↪'NUM', 'X', 'DET', 'NOUN', '.']

```

1.4 Vocabulary

As always when dealing with NLP models, we need an object to maintain our vocabulary of known tokens. We will rely on word-tokenization.

Our vocabulary class will maintain the set of known tokens, and a dictionary to convert tokens to codes and codes back to tokens. The class will also take care of some special symbols (e.g., BOS, EOS, UNK, PAD).

Finally, if later on you test your model on sentences that are not word tokenized, you can use `nlt.tokenize.word_tokenize` or any other tokenizer you like (as long as the level of tokenization is similar to the one you used for training your model).

This class will be used for maintaining both the vocabulary of known tokens and the set of known tags for the tagger data.

```
[22]: import numpy as np
from itertools import chain
from collections import Counter, OrderedDict

class Vocab:

    def __init__(self, corpus: list, min_freq=1):
        """
        corpus: list of documents, each document is a list of tokens, each token is a string
        min_freq: words that occur less than this value are discarded
        """

        # Count word occurrences
        counter = Counter(chain(*corpus))
```

(continues on next page)

(continued from previous page)

```

# Sort them by frequency
sorted_by_freq_tuples = sorted(counter.items(), key=lambda pair: pair[1],  

reverse=True)

# Special tokens
self.pad_token = "-PAD-" # used to fill sequences in a batch to maximum sequence  

length
self.bos_token = "-BOS-" # begin of sequence
self.eos_token = "-EOS-" # end of sequence
self.unk_token = "-UNK-" # unknown symbol
self.pad_id = 0
self.bos_id = 1
self.eos_id = 2
self.unk_id = 3

self.known_symbols = [self.pad_token, self.bos_token, self.eos_token, self.unk_  

token]
self.counts = [0, 0]

# Vocabulary
self.word2id = OrderedDict()
self.word2id[self.pad_token] = self.pad_id
self.word2id[self.bos_token] = self.bos_id
self.word2id[self.eos_token] = self.eos_id
self.word2id[self.unk_token] = self.unk_id
self.min_freq = min_freq
for w, n in sorted_by_freq_tuples:
    if n >= min_freq: # discard infrequent words
        self.word2id[w] = len(self.known_symbols)
        self.known_symbols.append(w)
        self.counts.append(n)

# Store the counts as well
self.counts = np.array(self.counts)

def __len__(self):
    return len(self.known_symbols)

def __getitem__(self, word: str):
    """
    Return the id (int) of a word (str)
    """
    return self.word2id.get(word, self.unk_id)

def encode(self, doc: list, add_bos=False, add_eos=False, pad_right=0):
    """
    Transform a document into a numpy array of integer token identifiers.
    doc: list of tokens, each token is a string
    add_bos: whether to add the BOS token
    add_eos: whether to add the EOS token
    pad_right: number of suffix padding tokens
    """

```

(continues on next page)

(continued from previous page)

```

Return: a list of codes (possibly with BOS and EOS added as well as padding)
"""
    return [self.word2id.get(w, self.unk_id) for w in chain([self.bos_token] *_
→int(add_bos), doc, [self.eos_token] * int(add_eos), [self.pad_token] * pad_right)]

def batch_encode(self, docs: list, add_bos=False, add_eos=False):
    """
    Transform a batch of documents into a numpy array of integer token identifiers.
    This will pad the shorter documents to the length of the longest document.
    docs: a list of documents
    add_bos: whether to add the BOS token
    add_eos: whether to add the EOS token
    pad_right: number of suffix padding tokens

    Return: numpy array with shape [len(docs), longest_doc + add_bos + add_eos]
    """
    max_len = max(len(doc) for doc in docs)
    return np.array([self.encode(doc, add_bos=add_bos, add_eos=add_eos, pad_*
→right=max_len-len(doc)) for doc in docs])

def decode(self, ids, strip_pad=False):
    """
    Transform a np.array document into a list of tokens.
    ids: np.array with shape [num_tokens]
    strip_pad: whether PAD tokens should be deleted from the output

    Return: list of strings with size [num_tokens - num_padding]
    """
    if strip_pad:
        return [self.known_symbols[id] for id in ids if id != self.pad_id]
    else:
        return [self.known_symbols[id] for id in ids]

def batch_decode(self, docs, strip_pad=False):
    """
    Transform a np.array collection of documents into a collection of lists of_
→tokens.
    ids: np.array with shape [num_docs, max_length]
    strip_pad: whether PAD tokens should be deleted from the output

    Return: list of documents, each a list of tokens, each token a string
    """
    return [self.decode(doc, strip_pad=strip_pad) for doc in docs]

```

Let's see how this works on the tagger corpus:

```
[23]: # We get a vocabulary for words
word_vocab = Vocab>tagger_training_x, min_freq=2)
# and a vocabulary for tags
tag_vocab = Vocab>tagger_training_y, min_freq=1)
# You can see their sizes V and C:
```

(continues on next page)

(continued from previous page)

```
[len(word_vocab), len(tag_vocab)]
[23]: (3358, 16)
```

The `encode` method turns a sequence of (str) symbols into a sequence of (int) codes:

We can also have `encode` add some special symbols for us (but remember to be consistent, you should always have token sequences and tag sequences that match in length):

We can also encode and decode entire batches of sequences. This will use pad symbols/codes to make the sequences in the same batch have the same length:

```
[24]: word_vocab.batch_encode>tagger_training_x[:3], add_bos=False, add_eos=True)
```

```
[24]: array([[ 45, 907, 13, 36, 18, 600, 1078, 8, 1651, 1652, 6,
           41, 19, 36, 66, 71, 2194, 55, 5, 2195, 10, 5,
           487, 4, 21, 2, 0, 0, 0],
          [ 488, 14, 1309, 156, 2196, 3, 3, 16, 3, 31, 449,
           908, 19, 3, 601, 909, 772, 8, 343, 910, 26, 9,
           2197, 344, 4, 2, 0, 0, 0],
          [ 5, 542, 129, 40, 2198, 12, 34, 5, 1310, 264, 16,
           44, 378, 54, 37, 44, 324, 10, 5, 2199, 186, 8,
           25, 1311, 23, 9, 111, 4, 2]])
```

```
[25]: word_vocab.batch_decode(word_vocab.batch_encode>tagger_training_x[:3], add_bos=False, ↴
                           ↵add_eos=True), strip_pad=True)
```

```
[25]: [['they',
  'know',
  '!',
  'he',
  'is',
  'generally',
  'opposed',
  'to',
  'cop-killer',
  'bullets',
  ',',
  'but',
  'that',
  'he',
  'had',
  'some',
  'reservations',
  'about',
  'the',
  'language',
  'in',
  'the',
  'legislation',
  '.',
  '',
  '-EOS-'],
 ['california',
```

(continues on next page)

(continued from previous page)

```

"'s",
'education',
'department',
'suspects',
'-UNK-',
'-UNK-',
'for',
'-UNK-',
'at',
'40',
'schools',
'that',
'-UNK-',
'changed',
'wrong',
'answers',
'to',
'right',
'ones',
'on',
'a',
'statewide',
'test',
'.',
'-EOS-'],
['the',
'loan',
'may',
'be',
'extended',
'*-1',
'by',
'the',
'mcalpine',
'group',
'for',
'an',
'additional',
'year',
'with',
'an',
'increase',
'in',
'the',
'conversion',
'price',
'to',
'$',
'2.50',
'*u*',
'a',
'share',

```

(continues on next page)

(continued from previous page)

```
'.',
'-EOS-']]
```

1.5 Corpus and Data Loader

We will be developing our models in torch, thus we need to wrap our corpus into a `Dataset` and a `DataLoader`:

```
[26]: import torch
from torch.utils.data import Dataset, DataLoader

class TextRegressionCorpus(Dataset):
    """
        Use this to give torch access to a corpus of documents annotated with a simple
        response variable (e.g., category or real number)
        This class will also know the vocab objects for tokens
        and it will take care of coding strings into integers consistently.
    """

    def __init__(self, corpus_x, corpus_y, vocab_x: Vocab):
        """
            In PyTorch we better always manipulate numerical codes, rather than text.
            So, our Corpus object will contain a vocab that converts words to codes.

            corpus_x: token sequences
            corpus_y: response values
            vocab_x: vocabulary of input symbols
        """

        self.corpus_x = list(corpus_x)
        self.corpus_y = list(corpus_y)
        if len(self.corpus_x) != len(self.corpus_y):
            raise ValueError("I need pairs")
        self.vocab_x = vocab_x

    def __len__(self):
        """Size of the corpus in number of sequence pairs"""
        return len(self.corpus_x)

    def __getitem__(self, idx):
        """Return corpus_x[idx] and corpus_y[idx] converted to codes and with the EOS
        code in the end"""
        x = self.vocab_x.encode(self.corpus_x[idx], add_bos=False, add_eos=True)
        y = self.corpus_y[idx]
        return x, y

    def pad_to_longest(self, pairs, pad_id=0):
        """
            Take a list of coded sequences and returns a torch tensor where
            every sentence has the same length (by means of using PAD tokens)
        """

        longest = max(len(x) for x, y in pairs)
        batch_x = torch.tensor([x + [self.vocab_x.pad_id] * (longest - len(x)) for x, y
                               in pairs])
```

(continues on next page)

(continued from previous page)

```

batch_y = torch.tensor([y for x, y in pairs])
return batch_x, batch_y

class ParallelCorpus(Dataset):
    """
    Use this to give torch access to a corpus of sequence pairs.
    This class will also know the vocab objects for the two streams.
    and it will take care of coding strings into integers consistently.
    """

    def __init__(self, corpus_x, corpus_y, vocab_x: Vocab, vocab_y: Vocab):
        """
        In PyTorch we better always manipulate numerical codes, rather than text.
        So, our Corpus object will contain a vocab that converts words to codes.

        corpus_x: token sequences
        corpus_y: tag sequences
        vocab_x: vocabulary for token sequences
        vocab_y: vocabulary for tag sequences
        """
        self.corpus_x = list(corpus_x)
        self.corpus_y = list(corpus_y)
        assert len(self.corpus_x) == len(self.corpus_y), "I need sequence pairs"
        self.vocab_x = vocab_x
        self.vocab_y = vocab_y

    def __len__():
        """Size of the corpus in number of sequence pairs"""
        return len(self.corpus_x)

    def __getitem__(idx):
        """
        Return corpus_x[idx] and corpus_y[idx] converted to codes
        the latter has the EOS code in the end
        """
        x = self.vocab_x.encode(self.corpus_x[idx], add_bos=False, add_eos=True)
        y = self.vocab_y.encode(self.corpus_y[idx], add_bos=False, add_eos=True)
        return x, y

    def pad_to_longest(pairs, pad_id=0):
        """
        Take a list of coded sequences and returns a torch tensor where
        every sentence has the same length (by means of using PAD tokens)
        """
        longest_x = max(len(x) for x, y in pairs)
        longest_y = max(len(y) for x, y in pairs)
        batch_x = torch.tensor([x + [self.vocab_x.pad_id] * (longest_x - len(x)) for x, _ in pairs])
        batch_y = torch.tensor([y + [self.vocab_y.pad_id] * (longest_y - len(y)) for _, y in pairs])
        return batch_x, batch_y

```

(continues on next page)

(continued from previous page)

```

class TaggedCorpus(ParallelCorpus):
    """
    Use this to give torch access to a corpus of tagged sequences.
    This class will also know the vocab objects for tokens and tags,
    and it will take care of coding strings into integers consistently.
    """

    def __init__(self, corpus_x, corpus_y, vocab_x: Vocab, vocab_y: Vocab):
        """
        In PyTorch we better always manipulate numerical codes, rather than text.
        So, our Corpus object will contain a vocab that converts words to codes.

        corpus_x: token sequences
        corpus_y: tag sequences
        vocab_x: vocabulary for token sequences
        vocab_y: vocabulary for tag sequences
        """
        super().__init__(corpus_x, corpus_y, vocab_x, vocab_y)
        assert all(len(x) == len(y) for x, y in zip(corpus_x, corpus_y)), "A sequence
        pair should match in number of steps"

```

We join our input and output data into torch Dataset objects for training, development and testing. Note that training, development and testing data share the same vocabularies, which were constructed using the training set alone. We can wrap the tagger data around a Dataset object as:

```
[27]: tagger_training = TaggedCorpus(tagger_training_x, tagger_training_y, word_vocab, tag_
    ↵vocab)
```

Here's an example of how we get a DataLoader for a corpus, we simply choose the Dataset object we want (training/dev/test), the batch size we want, whether we need shuffling (e.g., for training batches in SGD), and how we "glue" data points of different length together (i.e., a function such as pad_to_longest which TextRegressionCorpus and TaggedCorpus provide for us).

```
[28]: batcher = DataLoader(tagger_training, batch_size=3, shuffle=True, collate_fn>tagger_-
    ↵training.pad_to_longest)
for batch_x, batch_y in batcher:
    print("# This is how the tagged sequences in a batch come out of the data loader\n")
    for x, y in zip(batch_x, batch_y):
        print(x)
        print(y)
        print()

    print("# And we can always decode them for inspection\n")
    # stripping padding makes it easier to read the examples
    for x, y in zip(word_vocab.batch_decode(batch_x, strip_pad=True), tag_vocab.batch_-
        ↵decode(batch_y, strip_pad=True)):
        print(x)
        print(y)
        print()
    break
```

```
# This is how the tagged sequences in a batch come out of the data loader

tensor([ 77, 2535,      5,   318,      6,    43,   776,      4,      2,      0,      0,
        0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0])
tensor([14,   5,     8,     4,     6,   14,     4,     6,     2,     0,     0,     0,     0,     0,     0,
        0,     0,     0,     0,     0,     0])
tensor([  3,     3,    11,     3,   98,    13,    45,    29,     3,     7,   101,   918,
        175,   34,  1491,   15,     8,     3,    49,     3,   748,   213,     4,     2])
tensor([ 4,     4,   15,     4,     5,   10,   14,     5,     9,     7,     8,     4,   10,     7,     4,   10,   13,     5,
        15,     5,     4,     4,     6,     2])
tensor([ 46,  882,    69,     2,     0,     0,     0,     0,     0,     0,     0,     0,     0,
        0,     0,     0,     0,     0,     0,     0,     0])
tensor([4, 4, 6, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

# And we can always decode them for inspection

['i', 'loved', 'the', 'school', ',', 'its', 'history', '.', '-EOS-']
['PRON', 'VERB', 'DET', 'NOUN', '.', 'PRON', 'NOUN', '.', '-EOS-']

['-UNK-', '-UNK-', 'and', '-UNK-', 'say', '0', 'they', 'are', '-UNK-', 'of', 'any',
 ← 'efforts', '*ich*-1', 'by', 'mcgraw-hill', '*', 'to', '-UNK-', 'or', '-UNK-', 'scoring
 ← ', 'high', '.', '-EOS-']
['NOUN', 'NOUN', 'CONJ', 'NOUN', 'VERB', 'X', 'PRON', 'VERB', 'ADJ', 'ADP', 'DET', 'NOUN
 ← ', 'X', 'ADP', 'NOUN', 'X', 'PRT', 'VERB', 'CONJ', 'VERB', 'NOUN', 'NOUN', '.', '-EOS-
 ← ']

['new', 'account', ':', '-EOS-']
['NOUN', 'NOUN', '.', '-EOS-']
```

4.32.3 2. Text encoders

In NLP applications, we often have to *encode* a piece of text. For example, that is the case in text classification and regression, as well as in sequence labelling.

Assume for example, a text classifier that takes a document $x_{1:l} = \langle x_1, \dots, x_l \rangle$, where each token $x_i \in \mathcal{W}$ is from a finite vocabulary of V tokens and predicts a distribution over C classes from a set $\mathcal{T} = \{1, \dots, C\}$. An encoding function can map $x_{1:l}$ to a D -dimensional vector \mathbf{u} , which we can then transform to a C -dimensional vector of scores using an affine transformation, which in turn we can constrain to the probability simplex via softmax:

$$Y|X_{1:l} = x_{1:l} \sim \text{Categorical}(\mathbf{g}(x_{1:l}; \theta)) \quad (4.19)$$

$$\mathbf{u} = \text{encode}_D(x_{1:l}; \theta_{\text{enc}}) \quad (4.20)$$

$$\mathbf{s} = \text{affine}_C(\mathbf{u}; \theta_{\text{out}}) \quad (4.21)$$

$$\mathbf{g}(x_{1:l}; \theta) = \text{softmax}(\mathbf{s}) \quad (4.22)$$

Here I use a subscript to indicate the dimensionality of the output of the function, the named parameters after ; are the trainable parameters of the function.

Generally text encoders may return a single output vector per document, or one vector per word in the document.

```
[29]: import torch
import torch.nn as nn
import torch.nn.functional as F

class Encoder(nn.Module):

    def __init__(self, output_dim):
        super().__init__()
        self._output_dim = output_dim

    @property
    def output_dim(self):
        return self._output_dim

    def forward(self, x):
        """
        x: [batch_size, max_length]

        Return a tensor of shape [batch_size, max_length, output_dim] or
        [batch_size, output_dim]
        """
        raise NotImplementedError("Implement me!")
```

We develop a very basic text encoder based on a bidirectional LSTM. In principle, any powerful encoder (including pretrained encoders) can be used, we go ahead with the BiLSTM to keep the tutorial lightweight.

```
[30]: class TextEncoder(Encoder):
    """
    Given a predictor x this NN parameterises the pdf of the random variable Y|X=x.
    In other words, it predicts the conditional distribution P(Y|X=x).
    When reduce_mean is True the forward will return a single output vector per document.
    When reduce_mean is False one vector per word in the document is returned instead.
    """

    def __init__(self, vocab_size: int, word_embed_dim: int, hidden_size: int, reduce_mean=False, pad_id=0, p_drop=0.):
        super().__init__(2 * hidden_size)
        self.pad_id = pad_id
        self.word_embed_dim = word_embed_dim
        self.hidden_size = hidden_size
        self.vocab_size = vocab_size
        self.word_embed = nn.Embedding(self.vocab_size, embedding_dim=word_embed_dim)
        self.encoder = nn.LSTM(
            input_size=word_embed_dim,
            hidden_size=hidden_size,
            num_layers=1,
            batch_first=True,
            bidirectional=True,
        )
        self.reduce_mean = reduce_mean

    def forward(self, x):
```

(continues on next page)

(continued from previous page)

```

"""
x: [batch_size, max_length]
"""

# We begin by embedding the tokens
# [batch_size, max_length, embed_dim]
h = self.word_embed(x)
# [batch_size, max_length, 2*hidden_size*num_layers]
h, _ = self.encoder(h)
if self.reduce_mean: # average pooling
    h = torch.where((x == self.pad_id).unsqueeze(-1), torch.zeros_like(h), h)
    h = torch.sum(h, axis=-2) / torch.sum((x != self.pad_id).float(), -1, ↴
keepdim=True)
return h

```

4.32.4 3. Probabilistic model

A probabilistic model prescribes the probability measure of a random experiment, in this tutorial we will design models by explicitly parameterising a probability density function (pdf) using NNs. When our random variables (rvs) are multivariate or structured we will pick a factorisation of the joint pdf, decide on the statistical family of each factor, and parameterise the factors using NNs.

Generically, we will be modelling the distribution of some random variable Y (univariate, multivariate, structured) conditioned on an assignment of some random variable $X = x$ (typically structured).

For example, x may be a piece of text or an image. The response variable may be a category, a numerical measurement, a vector of attributes/measurements, a data structure (e.g., sequence, tree, graph).

For us, a probabilistic model can at the very least * assign probability density to an assignment $Y = y$ given $X = x$
* sample an assignment of Y given $X = x$ and we will use the forward method of a PyTorch Module to parameterise the relevant conditional probability distributions.

Occasionally, we will be able to support other operations such as finding the median, mean, mode, etc.

```
[31]: class Model(nn.Module):
    """
    A probabilistic model predicts the probability measure of a random experiment.
    We will predict the distribution of the random variable Y conditioned on
    an assignment to the random variable X=x.

    As a modelling mechanism we will use pmfs and pdfs, hence predicting
    the distribution  $P(Y|X=x)$  requires mapping from  $x$  to the parameters of a
    pmf/pdf for  $Y$ , or, in case of multivariate/structured data, a joint pdf
    whose factorisation we decide upon.

    For us, a model needs to satisfy the following desiderata:
    * parameterising the pdf is tractable
    * assessing the pdf for a given outcome  $Y=y|X=x$  is tractable
    * drawing samples from  $P(Y|X=x)$  is tractable
    * the pdf  $p(y|x)$  is differentiable with respect to the parameters of the
      NN that predicts its parameter(s)
    """


```

The only aspects of the Model's API that is fixed are:

(continues on next page)

(continued from previous page)

```

* log_prob(x, y) -> tensor of floats, one value per instance in the batch
* sample(x, sample_size) -> batch of samples
* forward(...) -> a torch.distribution object
  the signature of the forward method may vary in subclasses

"""

def __init__(self, event_shape=tuple()):
    """
    The event_shape is the shape of the outcome of the rv Y.
    """
    super().__init__()
    self._event_shape = event_shape

@property
def event_shape(self):
    return self._event_shape

def num_parameters(self):
    return sum(np.prod(theta.shape) for theta in self.parameters())

def forward(self, x):
    """
    x: [batch_size, ...]
    """
    raise NotImplementedError("Each type of model will have a different implementation here")

def sample(self, x, sample_size=tuple()):
    """
    x: [batch_size, ...]
    Return a batch of samples from Y/X=x
    """
    raise NotImplementedError("Each type of model will have a different implementation here")

def log_prob(self, x, y):
    """
    Computes the log pdf for Y=y/X=x for each pair in the batch.

    x: batch_shape + event_shape_x
    y: batch_shape + event_shape
    """

    # Predict the conditional probability distribution using the forward function.
    # This will return one such probability distribution per batch element.
    cpds = self(x=x)

    # Computer the log probability of each element in the batch.
    logp = cpds.log_prob(y) # [batch_size]

    return logp

```

4.32.5 4. Parameter estimation

We will estimate parameters using maximum likelihood estimation (MLE), via gradient-based search. This means we need to assess the model's likelihood given a dataset (or batch) of observations and the likelihood function must be tractable and differentiable with respect to the NN parameters.

```
[32]: def loss(self, x, y):
    """
    No matter the probabilistic model, the loss is the negative log likelihood
    of the parameters estimated on a single batch:
    - 1/batch_size * \sum_{s} log P(y[s]|x[s], theta)

    x: batch_shape + event_shape_x
    y: batch_shape + event_shape
    """
    return -self.log_prob(x=x, y=y).mean(0)
```

```
[33]: def distortion(model, dl, device):
    """
    Wrapper for estimating distortion using all data points in a data loader.
    """

    total_log_prob = 0.
    data_size = 0
    with torch.no_grad():
        for batch_x, batch_y in dl:
            total_log_prob = total_log_prob + model.log_prob(batch_x.to(device), batch_y.
            →to(device)).sum()
            data_size += batch_x.shape[0]
    return -total_log_prob / data_size
```

4.32.6 5. Decision rules

A rational decision maker chooses her actions to maximise expected utility.

Let $u(y, c)$ quantify the benefit in choosing $c \in \mathcal{Y}$ when y is the truth. When deciding under uncertainty, we solve

$$y^* = \arg \max_{c \in \mathcal{Y}} \mathbb{E}[u(Y, c)|X = x] \quad (4.23)$$

where the expectation is with respect to the pdf $p_{Y|X=x}$.

For certain utility functions and pdf combinations, this can be solved in closed form. In many cases there is not tractable algorithm for this decision problem, then approximations are needed.

When the utility function is $u(y, c) = [y = c]$ (which evaluates to 1 if y and c are the same, and 0 otherwise), y^* corresponds to the mode of $p_{Y|X=x}$.

In structure prediction, utility functions may reward partial/structural similarity between y and c . For example, a utility function for strings may be based on Levenshtein distance/similarity.

```
[34]: class DecisionRule:

    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```

def __call__(self, model, x):
    """
        This function should map from a model and input, and the resulting predicted
        probability distribution,
        to a single outcome.
    """
    raise NotImplementedError("Implement me!")

class ExactMode(DecisionRule):
    """
        This decision rule returns the most probable outcome under the predicted probability
        distribution, assuming
        a unimodal or discrete probability distribution.
    """

    def __call__(self, model, x):
        return model.mode(x)

```

We have some helper code for predictions using the batches in a data loader:

```
[35]: def predict(model, rule, dl, device, return_targets=False, strip_pad=True):
    """
        Wrapper for predictions using a decision rule.

        model: one of our taggers
        dl: a data loader for the heldout data
        device: the PyTorch device where the model is stored
        return_targets: also return the targets from the data loader
            you can use this when the actual targets are in the dataloader (e.g., for dev-
        ↵set)

        Return
            * a list of predictions, each a sequence of tags (already decoded)
            * if return_targets=True, additionally return a list of targets, each a sequence
        ↵of tags (already decoded)
    """

    all_preds = []
    all_targets = []
    with torch.no_grad():
        for batch_x, batch_y in dl:
            preds = rule(model, batch_x.to(device))
            all_preds.extend(preds.cpu().numpy())
            all_targets.extend(batch_y.cpu().numpy())

    if return_targets:
        return all_preds, all_targets
    else:
        return all_preds

```

4.32.7 6. Training procedure

The training procedure is always the exact same, no matter the model, so we abstract it for you.

```
[36]: import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.distributions as td
import torch.optim as opt
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
```

```
[37]: from sklearn.metrics import classification_report, mean_squared_error, mean_absolute_
error, median_absolute_error
from collections import defaultdict
from itertools import chain

def flatten(seq):
    """flattens a python list"""
    return list(chain.from_iterable(seq))

def report_regression(y_true, y_pred):
    return {"MSE": mean_squared_error(y_true, y_pred), "MAE": mean_absolute_error(y_true,
→ y_pred), "MdAE": median_absolute_error(y_true, y_pred)}

def report_classification(y_true, y_pred):
    return classification_report(y_true, y_pred, output_dict=True, zero_division=0)

def report_tagging(y_true, y_pred, score_pad=False, pad_id=0):
    if not score_pad:
        pairs = [(t, p) for t, p in zip(flatten(y_true), flatten(y_pred)) if t != pad_id]
        y_true = [t for t, p in pairs]
        y_pred = [p for t, p in pairs]
    return classification_report(y_true, y_pred, output_dict=True, zero_division=0)
```

```
[38]: def train_neural_model(model: Model, optimiser, decision_rule: DecisionRule,
                           training_data, dev_data,
                           batch_size=200, num_epochs=10, check_every=10,
                           report_fn=None, report_metrics=[],
                           device=torch.device('cuda:0'))
                           ):
    """
    model: pytorch model
    optimiser: pytorch optimiser
    training_corpus: a TaggedCorpus for trianing
    dev_corpus: a TaggedCorpus for dev
    batch_size: use more if you have more memory
    num_epochs: use more for improved convergence
    check_every: use less to check performance on dev set more often
    device: where we run the experiment
    """

    Return a log of quantities computed during training (for plotting)
```

(continues on next page)

(continued from previous page)

```

"""
# We use the training data in random order for parameter estimation
batcher = DataLoader(training_data, batch_size=batch_size, shuffle=True, collate_
fn=training_data.pad_to_longest)
# We use the dev data for evaluation during training (no need for randomisation here)
dev_batcher = DataLoader(dev_data, batch_size=batch_size, shuffle=False, collate_
fn=dev_data.pad_to_longest)

total_steps = num_epochs * len(batcher)
log = defaultdict(list)

model.eval()
log['D'].append(distortion(model, dev_batcher, device=device).item())

if report_fn:
    preds, targets = predict(
        model,
        decision_rule,
        dev_batcher,
        device=device,
        return_targets=True
    )
    report = report_fn(targets, preds)
    for metric in report_metrics:
        log[metric].append(report[metric])

step = 0

with tqdm(range(total_steps)) as bar:
    for epoch in range(num_epochs):
        for batch_x, batch_y in batcher:
            model.train()
            optimiser.zero_grad()

            L = loss(model, batch_x.to(device), batch_y.to(device))

            L.backward()
            optimiser.step()

            bar_dict = OrderedDict()
            bar_dict['loss'] = f"{L.item():.2f}"
            bar_dict['D'] = f'{log["D"][-1]:.2f}'
            for metric in report_metrics:
                bar_dict[metric] = f'{log[metric][-1]:.2f}'.format(log[metric][-1])
            bar.set_postfix(bar_dict)
            bar.update()

            log['loss'].append(L.item())

            if step % check_every == 0:
                model.eval()
                log['D'].append(distortion(model, dev_batcher, device=device).item())

```

(continues on next page)

(continued from previous page)

```

if report_fn:
    preds, targets = predict(
        model,
        decision_rule,
        dev_batcher,
        device=device,
        return_targets=True
    )
    report = report_fn(targets, preds)
    for metric in report_metrics:
        log[metric].append(report[metric])

    step += 1

model.eval()
log['D'].append(distortion(model, dev_batcher, device=device).item())

if report_fn:
    preds, targets = predict(
        model,
        decision_rule,
        dev_batcher,
        device=device,
        return_targets=True
    )
    report = report_fn(targets, preds)
    for metric in report_metrics:
        log[metric].append(report[metric])

return log

```

4.32.8 7. Examples

7.1 Categorical

Here we design a probabilistic model for a Categorical response variable Y given a document x :

$$Y|X=x \sim \text{Categorical}(\mathbf{g}(x; \theta)) \quad (4.24)$$

$$\mathbf{u} = \text{encode}_D(x; \theta_{\text{enc}}) \quad (4.25)$$

$$\mathbf{s} = \text{affine}_C(\mathbf{u}; \theta_{\text{out}}) \quad (4.26)$$

$$\mathbf{g}(x; \theta) = \text{softmax}(\mathbf{s}) \quad (4.27)$$

[39]: `class CategoricalModel(Model):`

`"""`

Given a predictor x this NN parameterises the pdf of the random variable $Y|X=x$, where Y is Categorically distributed.

In other words, it predicts the conditional distribution $P(Y|X=x)$.

(continues on next page)

(continued from previous page)

```

.....
def __init__(self, support_size, hidden_size: int, encoder: Encoder, p_drop=0.):
    super().__init__(tuple())
    self.encoder = encoder
    self.support_size = support_size
    self.hidden_size = hidden_size

    # We have a simple neural network that maps from the text encodings to
    # logits for the Categorical distribution.
    self.logits_predictor = nn.Sequential(
        nn.Dropout(p_drop),
        nn.Linear(encoder.output_dim, hidden_size),
        nn.ReLU(),
        nn.Dropout(p_drop),
        nn.Linear(hidden_size, support_size)
    )

def forward(self, x):

    # We begin by encoding the tokens
    h = self.encoder(x) # batch_shape, (enc_dim,)

    # We use the logit predictor network to transform those into logits.
    logits = self.logits_predictor(h) # batch_shape, (support_size,)

    # We return a Categorical distribution with the predicted logits as its
    # parameters.
    return td.Categorical(logits=logits)

def sample(self, x, sample_size=tuple()):
    with torch.no_grad():
        cpd = self(x=x)
        return cpd.sample(sample_size)

def mode(self, x):
    with torch.no_grad():
        # Predict the Categorical distribution of P(Y|X=x)
        cpd = self(x=x)

        # We can easily obtain the exact mode by taking the argmax over logits.
        mode = torch.argmax(cpd.logits, -1)

    return mode

```

Let's sanity check our implementation by doing a forwards pass through a randomly initialized text encoder and Categorical model:

```
[40]: encoder = TextEncoder(len(word_vocab), 7, 5, True)
cat_model = CategoricalModel(3, 12, encoder)
for batch_x, batch_y in batcher:
    print(batch_x.shape)
```

(continues on next page)

(continued from previous page)

```

print(cat_model(batch_x).logits)
print(cat_model.mode(batch_x))
print(cat_model.sample(batch_x))
break

torch.Size([3, 28])
tensor([[-1.2122, -1.1352, -0.9647],
       [-1.1917, -1.1365, -0.9799],
       [-1.2199, -1.1338, -0.9599]], grad_fn=<SubBackward0>)
tensor([2, 2, 2])
tensor([1, 2, 1])

```

[41]: # We already set up the data at the start of this notebook. We are only left to construct the word vocabulary.

```

cat_vocab = Vocab(cat_training_x, min_freq=2)
print(len(cat_vocab))

22734

```

[42]: # Now let's train a model on this data.

```

# First, reset random number generators
seed_all()

# Use GPU acceleration
my_device = torch.device('cuda:0')

# Create the model
model = CategoricalModel(
    support_size=len(brown.categories()),
    hidden_size=32,
    encoder=TextEncoder(
        vocab_size=len(cat_vocab),
        word_embed_dim=100,
        hidden_size=64,
        reduce_mean=True,
        pad_id=cat_vocab.pad_id
    ),
    p_drop=0.1
).to(my_device)

# Construct an Adam optimiser
optimiser = opt.Adam(model.parameters(), lr=5e-3)

print("Model")
print(model)

# Report number of parameters
print(f"Model size: {model.num_parameters():,} parameters")

# Train the model
log = train_neural_model(

```

(continues on next page)

(continued from previous page)

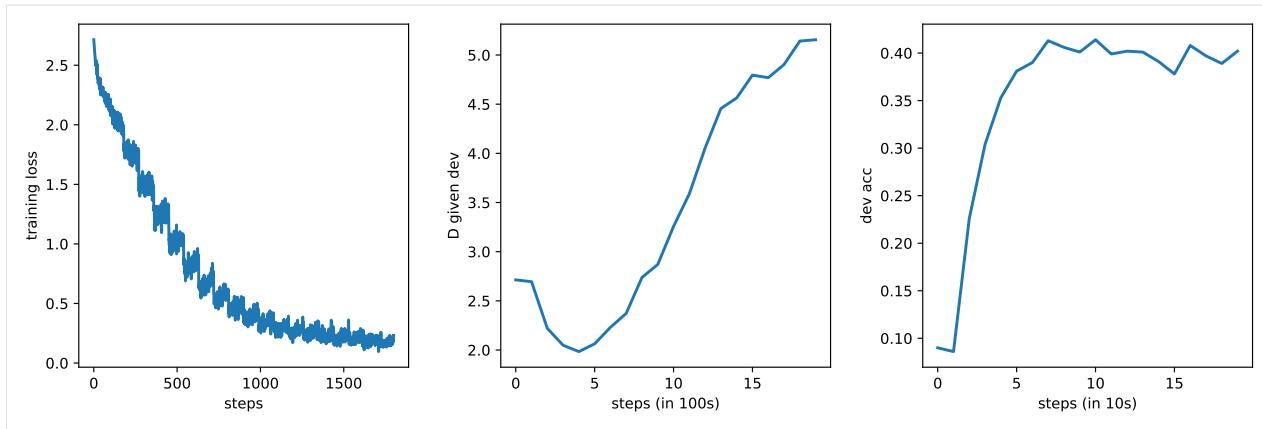
```

model, optimiser,
decision_rule=ExactMode(),
training_data=TextRegressionCorpus(cat_training_x, cat_training_y, cat_vocab),
dev_data=TextRegressionCorpus(cat_dev_x, cat_dev_y, cat_vocab),
report_fn=report_classification,
report_metrics=['accuracy'],
batch_size=500, num_epochs=20, check_every=100,
device=my_device
)

# Plot loss and validation checks
fig, axs = plt.subplots(1, 3, sharey=False, figsize=(12, 4))
_ = axs[0].plot(np.arange(len(log['loss'])), log['loss'])
_ = axs[0].set_xlabel('steps')
_ = axs[0].set_ylabel('training loss')
_ = axs[1].plot(np.arange(len(log['D'])), log['D'])
_ = axs[1].set_xlabel('steps (in 100s)')
_ = axs[1].set_ylabel('D given dev')
_ = axs[2].plot(np.arange(len(log['accuracy'])), log['accuracy'])
_ = axs[2].set_xlabel('steps (in 10s)')
_ = axs[2].set_ylabel('dev acc')
_ = fig.tight_layout(h_pad=2, w_pad=2)
plt.show()

Model
CategoricalModel(
    encoder: TextEncoder(
        (word_embed): Embedding(22734, 100)
        (encoder): LSTM(100, 64, batch_first=True, bidirectional=True)
    )
    (logits_predictor): Sequential(
        (0): Dropout(p=0.1, inplace=False)
        (1): Linear(in_features=128, out_features=32, bias=True)
        (2): ReLU()
        (3): Dropout(p=0.1, inplace=False)
        (4): Linear(in_features=32, out_features=15, bias=True)
    )
)
Model size: 2,363,015 parameters
0% | 0/1800 [00:00<?, ?it/s]

```



Let's use the model for some predictions.

```
[43]: # Set up the test dataloader
cat_test_corpus = TextRegressionCorpus(cat_test_x, cat_test_y, cat_vocab)
cat_test_dl = DataLoader(cat_test_corpus, batch_size=3, shuffle=True, collate_fn=cat_
    ↪_test_corpus.pad_to_longest)
batch_x, batch_y = next(iter(cat_test_dl))

# Use the mode to form predictions.
decision_rule = ExactMode()
predictions = decision_rule(model, batch_x.to(my_device))

for x, pred, truth in zip(batch_x, batch_y, predictions):
    print(f"input: {'.join(cat_vocab.decode(x, strip_pad=True))}")
    print(f"prediction: {brown.categories()[pred]}")
    print(f"truth: {brown.categories()[truth]}")
    print()
```

input: Have you examined this problem of increasing consumer sophistication from the
→standpoint of your own company ? ? -EOS-

prediction: hobbies

truth: lore

input: Some have plenty of money -- some have very little money . -EOS-

prediction: lore

truth: mystery

input: Thus far the advances made have been almost entirely along functional lines . -
→EOS-

prediction: belles_lettres

truth: learned

```
[44]: # We can also visualize the predicted Categoricals and use them for sampling or input
# to any arbitrary decision rule.
import seaborn as sns
import pandas as pd

with torch.no_grad():
```

(continues on next page)

(continued from previous page)

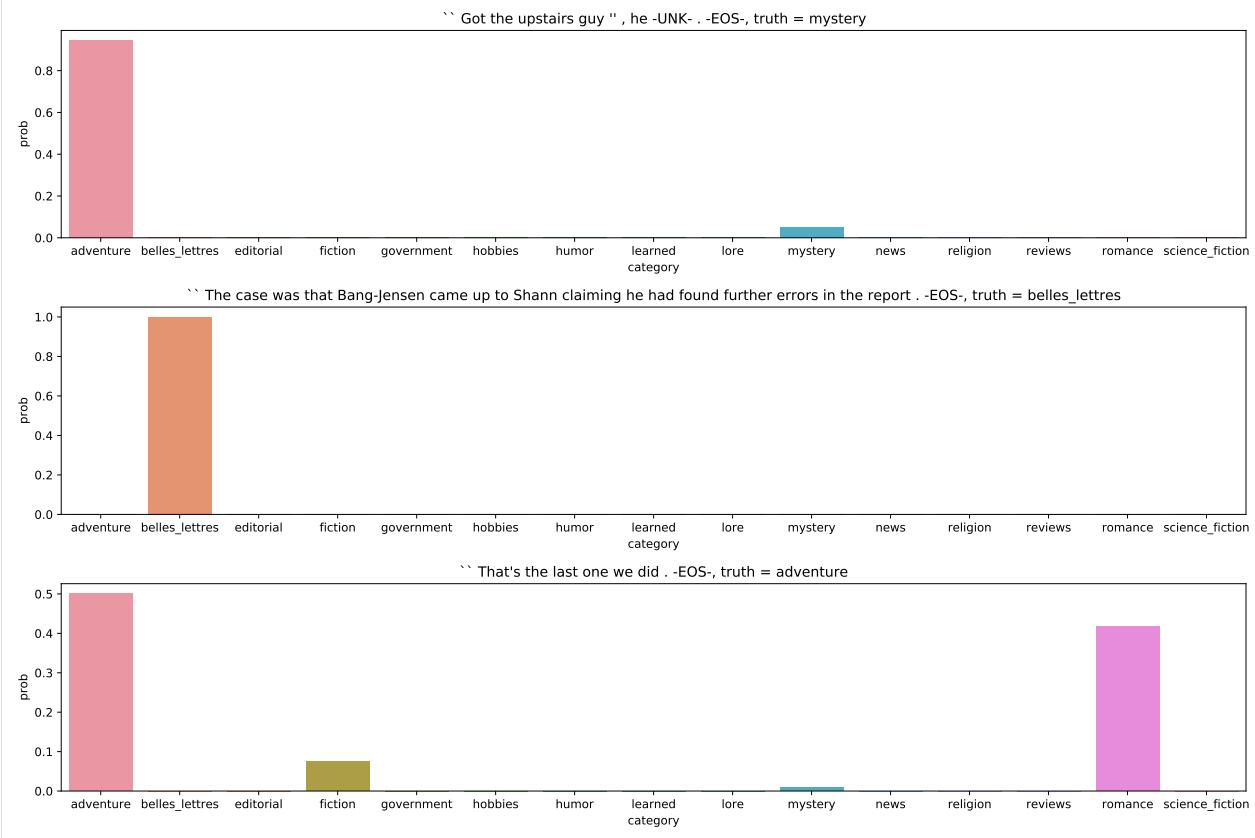
```

batch_x, batch_y = next(iter(cat_test_dl))
cpds = model(batch_x.to(my_device))

fig, axes = plt.subplots(3, 1, figsize=(15, 10))
for ax, x, y, predicted_probs in zip(axes, batch_x, batch_y, cpds.probs):
    sns.barplot(data=pd.DataFrame({"category": brown.categories(), "prob": predicted_probs.cpu()}),
                 x="category", y="prob", ax=ax)
    ax.set_title(f'{x.decode("utf-8")}, truth = {brown.categories()[y]}')

plt.tight_layout()
plt.show()

```



7.2 Poisson

Here we design a probabilistic model for an ordinal response variable Y given a document x , for which we choose to model with a Poisson pmf:

$$Y|X = x \sim \text{Poisson}(g(x; \theta)) \quad (4.28)$$

$$\mathbf{u} = \text{encode}_D(x; \theta_{\text{enc}}) \quad (4.29)$$

$$s = \text{affine}_1(\mathbf{u}; \theta_{\text{out}}) \quad (4.30)$$

$$g(x; \theta) = \text{softplus}(s) \quad (4.31)$$

The Poisson rate parameter must be strictly positive, thus we use a softplus output.

```
[45]: class PoissonModel(Model):
    """
    This model predicts a conditional Poisson distribution Y|X=x for ordinal data.
    """

    def __init__(self, hidden_size: int, encoder: Encoder, p_drop=0.):
        super().__init__(tuple())
        self.encoder = encoder
        self.hidden_size = hidden_size

        # The poisson distribution has 1 parameter: its rate. This needs to be a
        # positive real number.
        self.rate_predictor = nn.Sequential(
            nn.Dropout(p_drop),
            nn.Linear(encoder.output_dim, hidden_size),
            nn.ReLU(),
            nn.Dropout(p_drop),
            nn.Linear(hidden_size, 1),
            nn.Softplus()
        )

    def forward(self, x):
        h = self.encoder(x)
        rate = self.rate_predictor(h)
        return td.Poisson(rate=rate)

    def sample(self, x, sample_size=tuple()):
        with torch.no_grad():
            cpd = self(x=x)
            return cpd.sample(sample_size)

    def mode(self, x):
        with torch.no_grad():
            cpd = self(x=x)
            rate = cpd.rate
            return torch.floor(rate)
```

```
[46]: # Let's train on the age data using a conditional Poisson model.

# We need to construct the vocabulary for this data.
blog_vocab = Vocab(blog_training_x, min_freq=2)
print(f"len(cat_vocab): {len(cat_vocab)} words in the vocabulary.")

# Reset the random seeds
seed_all()

# Use GPU acceleration
my_device = torch.device('cuda:0')

# Create a similar model as before, but predicting a Poisson distribution instead.
model = PoissonModel(
    hidden_size=32,
```

(continues on next page)

(continued from previous page)

```

encoder=TextEncoder(
    vocab_size=len(blog_vocab),
    word_embed_dim=100,
    hidden_size=64,
    reduce_mean=True,
    pad_id=blog_vocab.pad_id
),
    p_drop=0.1
).to(my_device)

# construct an Adam optimiser
optimiser = opt.Adam(model.parameters(), lr=5e-3)

print("Model")
print(model)

# Report number of parameters
print(f"Model size: {model.num_parameters():,} parameters")

# Train the model
log = train_neural_model(
    model, optimiser,
    decision_rule=ExactMode(),
    training_data=TextRegressionCorpus(blog_training_x, blog_training_y, blog_vocab),
    dev_data=TextRegressionCorpus(blog_dev_x, blog_dev_y, blog_vocab),
    report_fn=report_regression,
    report_metrics=["MSE", "MAE", "MdAE"],
    batch_size=500, num_epochs=1, check_every=100,
    device=my_device
)

# Plot loss and validation checks
fig, axs = plt.subplots(1, 4, sharey=False, figsize=(16, 4))
_ = axs[0].plot(np.arange(len(log['loss'])), log['loss'])
_ = axs[0].set_xlabel('steps')
_ = axs[0].set_ylabel('training loss')
_ = axs[1].plot(np.arange(len(log['D'])), log['D'])
_ = axs[1].set_xlabel('steps (in 100s)')
_ = axs[1].set_ylabel('D given dev')
_ = axs[2].plot(np.arange(len(log['MAE'])), log['MAE'])
_ = axs[2].set_xlabel('steps (in 10s)')
_ = axs[2].set_ylabel('dev mean absolute error')
_ = axs[3].plot(np.arange(len(log['MdAE'])), log['MdAE'])
_ = axs[3].set_xlabel('steps (in 10s)')
_ = axs[3].set_ylabel('dev median absolute error')
_ = fig.tight_layout(h_pad=2, w_pad=2)
plt.show()

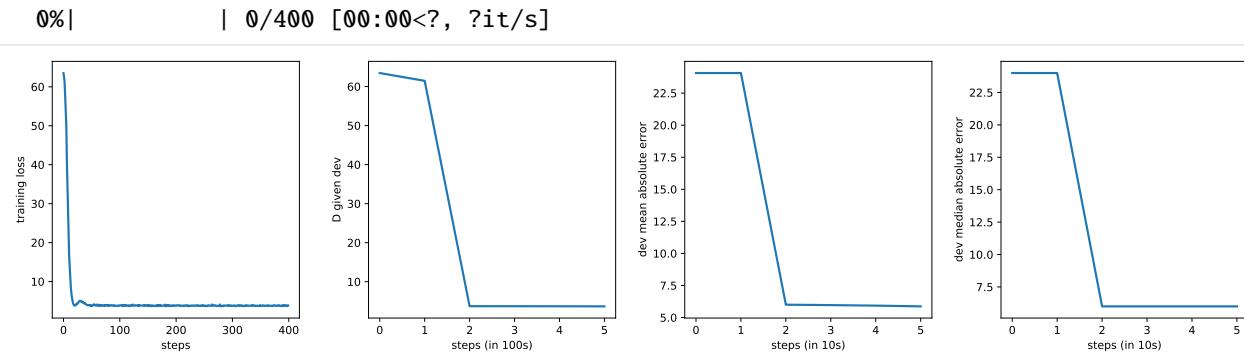
22,734 words in the vocabulary.
Model
PoissonModel(
    encoder): TextEncoder(

```

(continues on next page)

(continued from previous page)

```
(word_embed): Embedding(83532, 100)
(encoder): LSTM(100, 64, batch_first=True, bidirectional=True)
)
(rate_predictor): Sequential(
    (0): Dropout(p=0.1, inplace=False)
    (1): Linear(in_features=128, out_features=32, bias=True)
    (2): ReLU()
    (3): Dropout(p=0.1, inplace=False)
    (4): Linear(in_features=32, out_features=1, bias=True)
    (5): Softplus(beta=1, threshold=20)
)
)
Model size: 8,442,353 parameters
```



```
[47]: # Set up the test dataloader
blog_test_corpus = TextRegressionCorpus(blog_test_x, blog_test_y, blog_vocab)
blog_test_dl = DataLoader(blog_test_corpus, batch_size=3, shuffle=True, collate_fn=blog_
    ↪_test_corpus.pad_to_longest)
batch_x, batch_y = next(iter(blog_test_dl))

# We can use the mode again to form predictions.
decision_rule = ExactMode()
predictions = decision_rule(model, batch_x.to(my_device))

for x, pred, truth in zip(batch_x, batch_y, predictions):
    print(f"input: {' '.join(blog_vocab.decode(x, strip_pad=True))}")
    print(f"prediction: {pred.item()}")
    print(f"truth: {truth.item()}")
    print()

input: every day deserves a good urllink insult . but , no one insults like shakespeare_
    ↪.... : drunkenness is his best virtue , for he will be swine drunk , and in his sleep_
    ↪he does little harm , save to his -UNK- about him . -EOS-
prediction: 26
truth: 24.0

input: here 's something i never thought i 'd never see : an epa ad mocking the idea of_
    ↪good fuel -UNK- in cars . urllink http : -UNK- maybe the appliance department should_
    ↪talk to the transportation department and get everyone on the same page . -EOS-
prediction: 17
truth: 24.0
```

(continues on next page)

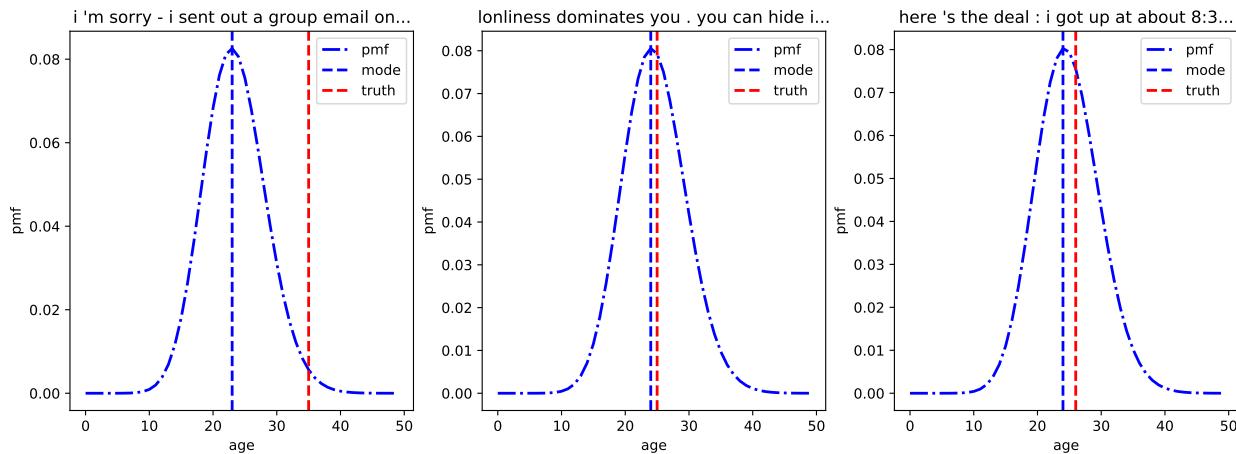
(continued from previous page)

```
input: i ca n't tolerate bitches ... im just allergic to them . they have no quality in_
→starting a conversation and worse of all , a disgrace to women around the world . its_
→in me ... i ca n't tolerate them . heard of a saying , respect yourself ... before_
→others respect you .. very the true ... -EOS-
prediction: 24
truth: 24.0
```

```
[48]: # We can also visualize the pmf of the predicted conditional Poisson distributions.
with torch.no_grad():
    batch_x, batch_y = next(iter(blog_test_dl))
    cpds = model(batch_x.to(my_device))
    mode = model.mode(batch_x.to(my_device))

    x = torch.arange(0, 50).repeat(3, 1)
    pmf = cpds.log_prob(x.to(my_device)).exp()

    fig, axes = plt.subplots(1, 3, figsize=(15, 5))
    for i in range(3):
        axes[i].plot(x[i].cpu(), pmf[i].cpu().numpy(), ls='-.', color='b', label='pmf')
        axes[i].axvline(mode[i].cpu().numpy(), ls='--', color='b', label='mode')
        axes[i].axvline(batch_y[i].numpy(), color='r', ls='--', label='truth')
        axes[i].set_xlabel("age")
        axes[i].set_ylabel("pmf")
        axes[i].set_title(f"{batch_x[i].cpu().numpy()[:40]}...")
    axes[i].legend()
```



7.3 Sequence labelling

In sequence labelling we have two sequences of equal length: a word sequence $x_{1:l}$ and a tag sequence $y_{1:l}$.

The word sequence $x_{1:l} = \langle x_1, \dots, x_l \rangle$, where l is the sequence length, is such that token x_i belongs to a vocabulary \mathcal{W} of V known words.

The tag sequence $y_{1:l} = \langle y_1, \dots, y_l \rangle$, where l is the same length as the document $x_{1:l}$, is such that each tag y_i belongs to a vocabulary \mathcal{T} of C known tags.

As always, our models are probability distributions, but this time we need a distribution over a space of sequences. A sequence is a finite length object, but it cannot be seen as a fixed dimensional vector (a multivariate random variable), for its length is not at all fixed. A sequence is best seen as a data structure. To prescribe a distribution over a general data structure, we can decompose this structure into parts, prescribe distributions for the parts, and combine them into a single pdf.

Chain rule allows us to re-express a pdf over an arbitrary sample space (with multivariate or structured outcomes) as a product of pdfs, each over a simpler space space (e.g., with univariate outcomes):

$$p(y_{1:l}|x_{1:l}, \theta) = \prod_{i=1}^l p(y_i|x_{1:l}, y_1^{i-1}, \theta) \quad (4.32)$$

$$= \prod_{i=1}^l \text{Categorical}(y_i|\mathbf{g}(x_{1:l}, y_1^{i-1}; \theta)) \quad (4.33)$$

where, on the right-hand side of the first equation, we have pdfs (or pmfs) for each step of the sequence. In the second row, we use a Categorical pmf since each step of the sequence is a simple categorical variable. Each pmf is parameterised in context using some trainable function \mathbf{g} with parameters θ .

The above is a *factorisation* of the joint pdf. This is one valid factorisation, and this one does not make any conditional independence assumptions (i.e., it conditions on all parts of the structure already generated). We could simplify the factorisation by making such conditional independence (or Markov) assumptions. For example, we might use only $x_{1:l}$ and the previous step y_{i-1} when parameterising the distribution of Y_i . These assumptions are justified on a case by case basis depending on application domain.

[49]: `class Tagger(Model):`

```

def __init__(self, vocab_size: int, tagset_size: int, pad_id=0, bos_id=1, eos_id=2):
    super().__init__()
    self._vocab_size = vocab_size
    self._tagset_size = tagset_size
    self._pad = pad_id
    self._bos = bos_id
    self._eos = eos_id

@property
def vocab_size(self):
    return self._vocab_size

@property
def tagset_size(self):
    return self._tagset_size

@property
def pad(self):
    return self._pad

```

(continues on next page)

(continued from previous page)

```

@property
def bos(self):
    return self._bos

@property
def eos(self):
    return self._eos

def num_parameters(self):
    return sum(np.prod(theta.shape) for theta in self.parameters())

def forward(self, x, y_in):
    """
    To predict the distribution of  $Y[i]$ , an autoregressive model conditions
    on both  $x$  and  $y[:i]$ . Normally,  $y_{in}$  is prepended with some BOS code so
    that both sequences have the same length.

    x: [batch_size, max_length]
    y_in: [batch_size, max_length]

    Return a batch of cpds, one per step.
    """
    raise NotImplementedError("Each type of tagger will have a different"
        "implementation here")

def log_prob(self, x, y):
    """
    Computes the log conditional probability of each tag sequence in a batch.

    x: [batch_size, max_length]
    y: [batch_size, max_length]
    """
    # shift the output sequence and prepend the BOS code
    # (after all, we do not want to condition on what we need to predict)
    batch_size, max_len = y.shape
    bos = torch.full((batch_size, 1), self.bos, device=y.device)
    y_in = torch.cat([bos, y], 1)[:, :-1]

    # one C-dimensional Categorical cpd for each token in the batch
    cpds = self(x=x, y_in=y_in)
    # [batch_size, max_length]
    logp = cpds.log_prob(y)
    # [batch_size]
    logp = torch.where(y != self.pad, logp, torch.zeros_like(logp)).sum(-1)
    return logp

def greedy(self, x):
    """
    For each cpd  $Y[i]/X=x$ , predicts the mode of the cpd.
    x: [batch_size, max_length]

```

(continues on next page)

(continued from previous page)

```

Return: tag sequences [batch_size, max_length]
"""
raise NotImplementedError("Each type of tagger differs here")

def sample(self, x, sample_size=tuple()):
    """
    Per word sequence in the batch, draws a number of samples from the model, each
    ↳sample is a complete tag sequence.

    x: [batch_size, max_len]

    Return: tag sequences with shape [batch_size, max_len] if sample_size is None
            else with shape [sample_size, batch_size, max_len]
    """
    raise NotImplementedError("Each type of tagger differs here")

```

```
[50]: # we get a vocabulary for words
word_vocab = Vocab>tagger_training_x, min_freq=2)
# and a vocabulary for tags
tag_vocab = Vocab>tagger_training_y, min_freq=1)
# you can see their sizes V and C:
len(word_vocab), len(tag_vocab)
```

```
[50]: (3358, 16)
```

7.3.1 Independent C-way classification

Our first tagger is in fact just a C -way classifier that we use to predict a distribution over C tags for different positions of an input sequence conditioned on the entire input sequence.

Here is the model of the i th tag given $x_{1:l}$:

$$Y_i | S = x_{1:l} \sim \text{Categorical}(\mathbf{g}(i, x_{1:l}; \theta)) \quad (4.34)$$

where \mathbf{g} is a neural network. For example:

$$\mathbf{e}_j = \text{embed}_D(x_j; \theta_{\text{in}}) \quad j \in \{1, \dots, l\} \quad (4.35)$$

$$\mathbf{u}_{1:l} = \text{birnn}_{2K}(\mathbf{e}_{1:l}; \theta_{\text{bienc}}) \quad (4.36)$$

$$\mathbf{s}_i = \text{affine}_C(\mathbf{u}_i; \theta_{\text{out}}) \quad (4.37)$$

$$\mathbf{g}(i, x_{1:l}) = \text{softmax}(\mathbf{s}_i) \quad (4.38)$$

The bidirection RNN layer concatenates the states of two independent RNN layers, one that processes the sequence from left-to-right, another that processes it from right-to-left.

Note how this model ignores every other tag in the sequence.

```
[51]: class BasicTagger(Tagger):

    def __init__(self, vocab_size, tagset_size, word_embed_dim: int, hidden_size: int,
                 drop=0, pad_id=0, bos_id=1, eos_id=2):
        """
        vocab_size: number of known words
```

(continues on next page)

(continued from previous page)

```

tagset_size: number of known tags
word_embed_dim: dimensionality of word embeddings
hidden_size: dimensionality of hidden layers
recurrent_encoder: enable recurrent encoder
bidirectional_encoder: for a recurrent encoder, make it bidirectional
"""

super().__init__(vocab_size=vocab_size, tagset_size=tagset_size, pad_id=pad_id, u
bos_id=bos_id, eos_id=eos_id)
    self.word_embed_dim = word_embed_dim
    self.hidden_size = hidden_size
    self.encoder = TextEncoder(
        vocab_size=vocab_size,
        word_embed_dim=word_embed_dim,
        hidden_size=hidden_size,
        reduce_mean=False,
        pad_id=pad_id,
        p_drop=p_drop
    )
    # the bidirectional LSTM encoder produces outputs of size 2*hidden_size
    # thus our linear layer must take 2*hidden_size inputs
    self.logits_predictor = nn.Sequential(
        nn.Dropout(p_drop),
        nn.Linear(self.encoder.output_dim, self.tagset_size)
    )

def forward(self, x, y_in=None):
    """
    Parameterise the conditional distributions over Y[i] given the entire word
    sequence x.

    x: [batch_size, max_length]
    y: not used by this class

    Return: a batch of C-dimensional Categorical distributions, one per step of the
    sequence.
    """
    # We begin by encoding the tokens
    # [batch_size, max_length, enc_dim]
    h = self.encoder(x)
    # finally, per step of the sequence we predict logits for the possible tags
    # [batch_size, max_length, tagset_size]
    s = self.logits_predictor(h)
    # and convert those logits to Categorical distributions
    return td.Categorical(logits=s)

def greedy(self, x):
    """
    For each cpd Y[i]|X=x, predicts the mode of the cpd.
    x: [batch_size, max_length]

    Return: tag sequences [batch_size, max_length]
    """

```

(continues on next page)

(continued from previous page)

```

batch_size = x.shape[0]
max_length = x.shape[1]
with torch.no_grad():
    cpds = self(x)
    # [batch_size, max_length]
    y_pred = torch.argmax(cpds.probs, -1)
    # if a position in x is padded, it should be padded in y
    y_pred = torch.where(x != self.pad, y_pred, torch.zeros_like(y_pred) + self.
    ↪pad)
return y_pred

def mode(self, x):
    return self.greedy(x)

def sample(self, x, sample_size=None):
    """
        Per word sequence in the batch, draws a number of samples from the model, each
        ↪sample is a complete tag sequence.

    x: [batch_size, max_len]

    Return: tag sequences with shape [batch_size, max_len] if sample_size is None
            else with shape [sample_size, batch_size, max_len]
    """
    batch_size = x.shape[0]
    max_length = x.shape[1]
    with torch.no_grad():
        cpds = self(x)
        if sample_size is None:
            shape = (1,)
        else:
            shape = (sample_size,)
        # [sample_size, batch_size, max_length]
        y_pred = cpds.sample(shape)
        # if a position in x is padding, it must be padded in y too
        y_pred = torch.where(x.unsqueeze(0) != self.pad, y_pred, torch.zeros_like(y_
    ↪pred) + self.pad)
        # takes care of output shape
        if sample_size is None:
            return y_pred.squeeze(0)
        else:
            return y_pred

```

```
[52]: def test_basic_tagger(training_x, training_y, vocab_x, vocab_y):
    seed_all()
    toy_uni_tagger = BasicTagger(
        vocab_size=len(vocab_x),
        tagset_size=len(vocab_y),
        word_embed_dim=32,
        hidden_size=32
    )
```

(continues on next page)

(continued from previous page)

```

assert type(toy_uni_tagger(torch.from_numpy(vocab_x.batch_encode(training_x[:2])))) is td.Categorical

assert toy_uni_tagger.log_prob(
    torch.from_numpy(vocab_x.batch_encode(training_x[:2])),
    torch.from_numpy(vocab_y.batch_encode(training_y[:2])))
).shape == (2,)

assert loss(toy_uni_tagger,
            torch.from_numpy(vocab_x.batch_encode(training_x[:2])),
            torch.from_numpy(vocab_y.batch_encode(training_y[:2])))
).shape == tuple()

assert toy_uni_tagger.sample(torch.from_numpy(vocab_x.batch_encode(training_x[:2]))).
shape == vocab_x.batch_encode(training_x[:2]).shape

assert toy_uni_tagger.sample(torch.from_numpy(vocab_x.batch_encode(training_x[:2])), 3).
shape == (3,) + vocab_x.batch_encode(training_x[:2]).shape

assert toy_uni_tagger.greedy(torch.from_numpy(vocab_x.batch_encode(training_x[:2]))).
shape == vocab_x.batch_encode(training_x[:2]).shape

test_basic_tagger>tagger_training_x, tagger_training_y, word_vocab, tag_vocab)

```

7.3.2 Autoregressive tagger

When predicting the distribution of Y_i , an autoregressive tagger conditions on the tag sequence already generated thus far, hence it makes no Markov assumption. This is the model of the i th tag:

$$Y_i|S = x_{1:l}, H = y_1^{i-1} \sim \text{Categorical}(g(i, x_{1:l}, y_1^{i-1}; \theta)) \quad (4.39)$$

where g is a neural network. For example:

$$\mathbf{e}_j = \text{embed}_{D_1}(x_j; \theta_{\text{words}}) \quad j \in \{1, \dots, l\} \quad (4.40)$$

$$\mathbf{t}_k = \text{embed}_{D_2}(y_k; \theta_{\text{tags}}) \quad k < i \quad (4.41)$$

$$\mathbf{u}_{1:l} = \text{birnn}_{2K}(\mathbf{e}_{1:l}; \theta_{\text{bienc}}) \quad (4.42)$$

$$\mathbf{v}_i = \text{rnnstep}_K(\mathbf{v}_{i-1}, \mathbf{t}_{i-1}; \theta_{\text{dec}}) \quad (4.43)$$

$$\mathbf{s}_i = \text{ffnn}_C(\text{concat}(\mathbf{u}_i, \mathbf{v}_i); \theta_{\text{out}}) \quad (4.44)$$

$$g(i, x_{1:l}) = \text{softmax}(\mathbf{s}_i) \quad (4.45)$$

Again, we have two different embedding layers, one for words and one for tags. Again, we use a bidirectional RNN to encode the whole document. Now, for the i th position, we use an RNN generator/decoder cell to encode the complete history of previous tags. We then predict the logits by using an FFNN to combine the history encoding with the document encoding for position i .

There's yet another way to parameterise this model, in which we let the RNN decoder compose the features of the

history with the features of the document:

$$\mathbf{e}_j = \text{embed}_{D_1}(x_j; \theta_{\text{words}}) \quad j \in \{1, \dots, l\} \quad (4.46)$$

$$\mathbf{t}_k = \text{embed}_{D_2}(y_k; \theta_{\text{tags}}) \quad k < i \quad (4.47)$$

$$\mathbf{u}_{1:l} = \text{birnn}_{2K}(\mathbf{e}_{1:l}; \theta_{\text{bienc}}) \quad (4.48)$$

$$\mathbf{v}_i = \text{rnnstep}_K(\mathbf{v}_{i-1}, \text{concat}(\mathbf{u}_i, \mathbf{t}_{i-1}); \theta_{\text{dec}}) \quad (4.49)$$

$$\mathbf{s}_i = \text{affine}_C(\mathbf{v}_i; \theta_{\text{out}}) \quad (4.50)$$

$$\mathbf{g}(i, x_{1:l}) = \text{softmax}(\mathbf{s}_i) \quad (4.51)$$

Both are very good options.

```
[53]: class AutoregressiveTagger(Tagger):

    def __init__(self, vocab_size: int, tagset_size: int, word_embed_dim: int, tag_embed_
        ↪dim: int, hidden_size: int, p_drop=0., pad_id=0, bos_id=1, eos_id=2):
        """
        ngram_size: longest ngram (for tag sequence)
        vocab_size: number of known words
        tagset_size: number of known tags
        word_embed_dim: dimensionality of word embeddings
        tag_embed_dim: dimensionality of tag embeddings (needed to encode the history of_
        ↪ngram_size-1 tags)
        hidden_size: dimensionality of hidden layers
        """
        super().__init__(vocab_size=vocab_size, tagset_size=tagset_size, pad_id=pad_id,_
        ↪bos_id=bos_id, eos_id=eos_id)
        self.word_embed_dim = word_embed_dim
        self.tag_embed_dim = tag_embed_dim
        self.hidden_size = hidden_size

        self.encoder = TextEncoder(
            vocab_size=vocab_size,
            word_embed_dim=word_embed_dim,
            hidden_size=hidden_size,
            reduce_mean=False,
            pad_id=pad_id
        )
        # we need to embed tags in the history
        self.tag_embed = nn.Embedding(tagset_size, embedding_dim=tag_embed_dim)

        self.decoder = nn.LSTM(
            input_size=tag_embed_dim,
            hidden_size=hidden_size,
            num_layers=1,
            batch_first=True,
            bidirectional=False
        )
        # for each position i, we need to combine the encoding of x[i] in context
        # as well as the history of ngram_size-1 tags
        # so we use a FFNN for that:
        self.logits_predictor = nn.Sequential(
            nn.Dropout(p_drop),
            nn.Linear(self.encoder.output_dim + hidden_size, hidden_size),
        )
```

(continues on next page)

(continued from previous page)

```

        nn.ReLU(),
        nn.Dropout(p_drop),
        nn.Linear(hidden_size, tagset_size),
    )

    self.init_state = nn.Sequential(
        nn.Dropout(p_drop),
        nn.Linear(self.encoder.output_dim, hidden_size),
        nn.Tanh()
)
    self.init_cell = nn.Sequential(
        nn.Dropout(p_drop),
        nn.Linear(self.encoder.output_dim, hidden_size),
        nn.Tanh()
)

def forward(self, x, y_in):
    """
    Parameterise the conditional distributions over Y[i] given history y[:i] and all
    of x.

    This procedure takes care that the ith output distribution conditions only on
    the n-1 observations before y[i].
    It also takes care of padding to the left with BOS symbols.

    x: word sequences [batch_size, max_length]
    y_in: history of tag sequences [batch_size, max_length]

    Return: a batch of V-dimensional Categorical distributions, one per step of the
    sequence.
    """

    # Let's start by encoding the conditioning sequences
    # [batch_size, max_length, enc_dim]
    u = self.encoder(x)

    # here we pad the tag sequence with BOS on the left
    # [batch_size, max_len, tag_emb_dim]
    t_in = self.tag_embed(y_in)

    # encode the histories
    # [batch_size, max_len, hidden_size]
    v, _ = self.decoder(t_in)

    # Now we can combine the encodings of x and the encodings of histories, we do so
    via concatenation
    # since there's a fixed number of such encodings per step of the sequence
    # [batch_size, max_length, 3*hidden_size]
    u = torch.cat([u, v], -1)
    # We are now ready to map the state of each step of the sequence to a C-
    dimensional vector of logits
    # we do so using our FFNN

```

(continues on next page)

(continued from previous page)

```

# [batch_size, max_length, tagset_size]
s = self.logits_predictor(u)

return td.Categorical(logits=s)

def greedy(self, x):
    """
    Draws a number of samples from the model, each sample is a complete sequence.
    We impose a maximum number of steps, to avoid infinite loops.
    This procedure takes care of mapping sampled symbols to pad after the EOS symbol
    is generated.
    """
    batch_size = x.shape[0]
    max_length = x.shape[1]

    with torch.no_grad():
        # add the beginning we do not know the tag sequence
        # but NNs work with fixed dimensional tensors,
        # so we allocate a tensor full of BOS codes
        y = torch.full((batch_size, max_length + 1), self.bos, device=self.tag_embed.
weight.device)
        # Per step
        for i in range(max_length):
            # we parameterise a cpd for Y[i]/X=x
            # note that the forward method takes care of not conditioning on y[i]_
itself
            # and only using the ngram_size-1 previous tags
            # at this point, the tag y[i] is a dummy code
            # the forward method recomputes all cts in the batch, this will include_
the cpd for Y[i]
            # [batch_size, max_len, C]
            cpds = self(x=x, y_in=y[:, :-1])
            # we get their modes via argmax
            # [batch_size, max_len]
            modes = torch.argmax(cpds.probs, -1)

            # Here we update the current token to the freshly obtained mode
            # and also replace the token by 0 (pad) in case the sentence is already_
complete
            y[:, i+1] = modes[:, i]
            # discard the BOS token
            y = y[:, 1:]
            # where we had a PAD token in x, we change the y token to PAD too
            y = torch.where(x != self.pad, y, torch.zeros_like(y) + self.pad)

    return y

def mode(self, x):
    raise NotImplementedError("The search for the mode of the autoregressive tagger
is intractable, consider using `greedy` or a sampling-based approximation. ")

def _sample(self, x):

```

(continues on next page)

(continued from previous page)

```

"""
Draws a number of samples from the model, each sample is a complete sequence.
We impose a maximum number of steps, to avoid infinite loops.
This procedure takes care of mapping sampled symbols to pad after the EOS symbol
→is generated.
"""

batch_size = x.shape[0]
max_length = x.shape[1]

with torch.no_grad():
    # add the beginning we do not know the tag sequence
    # but NNs work with fixed dimensional tensors,
    # so we allocate a tensor full of BOS codes
    y = torch.full((batch_size, max_length + 1), self.bos, device=self.tag_embed.
→weight.device)

    # Per step
    for i in range(max_length):
        # we parameterise a cpd for Y[i]|X=x
        # note that the forward method takes care of not conditioning on y[i]→
→itself
        # and only using the ngram_size-1 previous tags
        # at this point, the tag y[i] is a dummy code
        # the forward method recomputes all cds in the batch, this will include
→the cpd for Y[i]
        # we get their modes via argmax
        # [batch_size, max_len, C]
        cpds = self(x=x, y_in=y[:, :-1])
        # [batch_size, max_len]
        samples = cpds.sample()

        # Here we update the current token to the freshly obtained mode
        # and also replace the token by 0 (pad) in case the sentence is already
→complete
        y[:, i+1] = samples[:, i]
        # discard the BOS token
        y = y[:, 1:]
        # where we had a PAD token in x, we change the y token to PAD too
        y = torch.where(x != self.pad, y, torch.zeros_like(y) + self.pad)

    return y

def sample(self, x, sample_size=None):
    """
Draws a number of samples from the model, each sample is a complete sequence.
We impose a maximum number of steps, to avoid infinite loops.
This procedure takes care of mapping sampled symbols to pad after the EOS symbol
→is generated.
"""

    if sample_size is None:
        return self._sample(x)
    else:

```

(continues on next page)

(continued from previous page)

```
samples = [self._sample(x) for _ in range(sample_size)]
return torch.stack(samples)
```

```
[54]: def test_autoreg_tagger(training_x, training_y, vocab_x, vocab_y):
    seed_all()
    toy_ar_tagger = AutoregressiveTagger(
        vocab_size=len(vocab_x),
        tagset_size=len(vocab_y),
        word_embed_dim=32,
        tag_embed_dim=16,
        hidden_size=32
    )

    assert type(toy_ar_tagger) == td.Categorical

    assert toy_ar_tagger.log_prob(
        torch.from_numpy(vocab_x.batch_encode(training_x[:2])),
        torch.from_numpy(vocab_y.batch_encode(training_y[:2])))
    ) is tuple()

    assert loss(toy_ar_tagger,
        torch.from_numpy(vocab_x.batch_encode(training_x[:2])),
        torch.from_numpy(vocab_y.batch_encode(training_y[:2])))
    ).shape == tuple()

    assert toy_ar_tagger.sample(torch.from_numpy(vocab_x.batch_encode(training_x[:2]))).shape == vocab_x.batch_encode(training_x[:2]).shape

    assert toy_ar_tagger.sample(torch.from_numpy(vocab_x.batch_encode(training_x[:2])), 3).shape == (3,) + vocab_x.batch_encode(training_x[:2]).shape

    assert toy_ar_tagger.greedy(torch.from_numpy(vocab_x.batch_encode(training_x[:2]))).shape == vocab_x.batch_encode(training_x[:2]).shape

test_autoreg_tagger>tagger_training_x,>tagger_training_y,word_vocab,tag_vocab)
```

Because labelling is a chain of classification decisions, we can also evaluate our tagger in terms of accuracy of its decisions. For that we need a decision rule. Normally, in NLP, we use the most probable tag sequence as a decision. That is, given a sentence $x_{1:l}$ we search in the space $\{1, \dots, C\}^l$ of all tag sequences of length l , for the one sequence that the model assigns highest probability to (i.e., the *mode* of the conditional distribution):

$$y^* = \arg \max_{c_{1:l} \in \{1, \dots, C\}^l} \log P(G = c_{1:l} | S = x_{1:l}) \quad (4.52)$$

This search is defined over an extremely large space and is generally not tractable. For some types of tagger, because of their conditional independence assumptions, this search may be doable in polynomial time (as a function of sequence length), for others this is not at all possible.

For the basic tagger, which treats the tags as independent given the sentence, this search can be done exactly, because greedily maximising each step independently is equivalently to maximising the joint assignment of the entire sequence for that model.

Search for the unigram tagger

We search for the best tag in each position, which takes time $\mathcal{O}(C)$ per position,

$$y_i^* = \arg \max_{c \in \{1, \dots, C\}} \log P(Y_i = c | S = x_{1:l}) \quad (4.53)$$

and put them together in a sequence. The total operation takes time $\mathcal{O}(l \times C)$.

Search for the autoregressive tagger

The autoregressive tagger makes no conditional independence assumptions, and the search problem is genuinely intractable for this model. Being intractable means there is not efficient algorithm known to be able to handle it. In fact, the current hypothesis is that an efficient (by efficient we mean that it runs in polynomial time as a function of l) is actually impossible in standard computer architectures. Problems of this kind are called NP-complete.

For this tutorial, we will again use the greedy approximation:

$$\hat{y}_i = \arg \max_{c \in \{1, \dots, C\}} \log P(Y_i = c | S = x_{1:l}, H = \hat{y}_1^{i-1}) \quad (4.54)$$

where we solve the argmax locally per position in order from left-to-right. For each step Y_i we condition on the already predicted argmax for all the preceding steps.

Once we have a search algorithm in place to make predictions we can compute accuracy and/or other metrics common for classification.

```
[55]: class GreedyMode(DecisionRule):

    def __call__(self, model, x):
        return model.greedy(x)
```

On GPU, this should take just about 2 minutes:

```
[56]: seed_all() # reset random number generators before creating your model and training it

my_device = torch.device('cuda:0')

tagger = BasicTagger(
    vocab_size=len(word_vocab),
    tagset_size=len(tag_vocab),
    word_embed_dim=4,
    hidden_size=8,
).to(my_device)

# construct an Adam optimiser
optimiser = opt.Adam(tagger.parameters(), lr=5e-3)

print("Model")
print(tagger)

# report number of parameters
print(f"Model size: {tagger.num_parameters():,} parameters")

# Train the model
```

(continues on next page)

(continued from previous page)

```

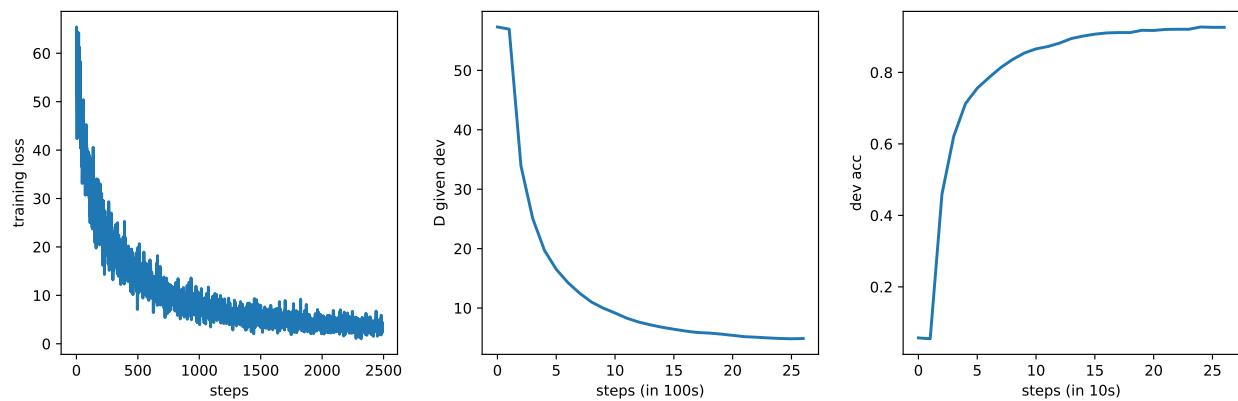
log = train_neural_model(
    tagger, optimiser,
    decision_rule=ExactMode(),
    training_data=TaggedCorpus(tagger_training_x, tagger_training_y, word_vocab, tag_
→vocab),
    dev_data=TaggedCorpus(tagger_dev_x, tagger_dev_y, word_vocab, tag_vocab),
    report_fn=report_tagging, report_metrics=['accuracy'],
    batch_size=10, num_epochs=10, check_every=100,
    device=my_device
)

# Plot loss and validation checks
fig, axs = plt.subplots(1, 3, figsize=(12, 4))
_ = axs[0].plot(np.arange(len(log['loss'])), log['loss'])
_ = axs[0].set_xlabel('steps')
_ = axs[0].set_ylabel('training loss')
_ = axs[1].plot(np.arange(len(log['D'])), log['D'])
_ = axs[1].set_xlabel('steps (in 100s)')
_ = axs[1].set_ylabel('D given dev')
_ = axs[2].plot(np.arange(len(log['accuracy'])), log['accuracy'])
_ = axs[2].set_xlabel('steps (in 10s)')
_ = axs[2].set_ylabel('dev acc')
_ = fig.tight_layout(h_pad=2, w_pad=2)
plt.show()

Model
BasicTagger(
    (encoder): TextEncoder(
        (word_embed): Embedding(3358, 4)
        (encoder): LSTM(4, 8, batch_first=True, bidirectional=True)
    )
    (logits_predictor): Sequential(
        (0): Dropout(p=0, inplace=False)
        (1): Linear(in_features=16, out_features=16, bias=True)
    )
)
Model size: 14,600 parameters

```

0% | 0/2490 [00:00<?, ?it/s]



```
[57]: seed_all() # reset random number generators before creating your model and training it

my_device = torch.device('cuda:0')

ar_tagger = AutoregressiveTagger(
    vocab_size=len(word_vocab),
    tagset_size=len(tag_vocab),
    word_embed_dim=4,
    tag_embed_dim=4,
    hidden_size=8,
).to(my_device)

# construct an Adam optimiser
optimiser = opt.Adam(ar_tagger.parameters(), lr=5e-3)

print("Model")
print(ar_tagger)

# report number of parameters
print(f"Model size: {ar_tagger.num_parameters():,} parameters")

# Train the model
log = train_neural_model(
    ar_tagger, optimiser,
    decision_rule=GreedyMode(),
    training_data=TaggedCorpus>tagger_training_x,>tagger_training_y,word_vocab,tag_vocab),
    dev_data=TaggedCorpus>tagger_dev_x,>tagger_dev_y,word_vocab,tag_vocab),
    report_fn=report_tagging, report_metrics=['accuracy'],
    batch_size=10, num_epochs=10, check_every=100,
    device=my_device
)

# Plot loss and validation checks
fig, axs = plt.subplots(1, 3, figsize=(12, 4))
_ = axs[0].plot(np.arange(len(log['loss'])), log['loss'])
_ = axs[0].set_xlabel('steps')
_ = axs[0].set_ylabel('training loss')
_ = axs[1].plot(np.arange(len(log['D'])), log['D'])
_ = axs[1].set_xlabel('steps (in 100s)')
_ = axs[1].set_ylabel('D given dev')
_ = axs[2].plot(np.arange(len(log['accuracy'])), log['accuracy'])
_ = axs[2].set_xlabel('steps (in 10s)')
_ = axs[2].set_ylabel('dev acc')
_ = fig.tight_layout(h_pad=2, w_pad=2)
plt.show()

Model
AutoregressiveTagger(
  (encoder): TextEncoder(
    (word_embed): Embedding(3358, 4)
    (encoder): LSTM(4, 8, batch_first=True, bidirectional=True)
```

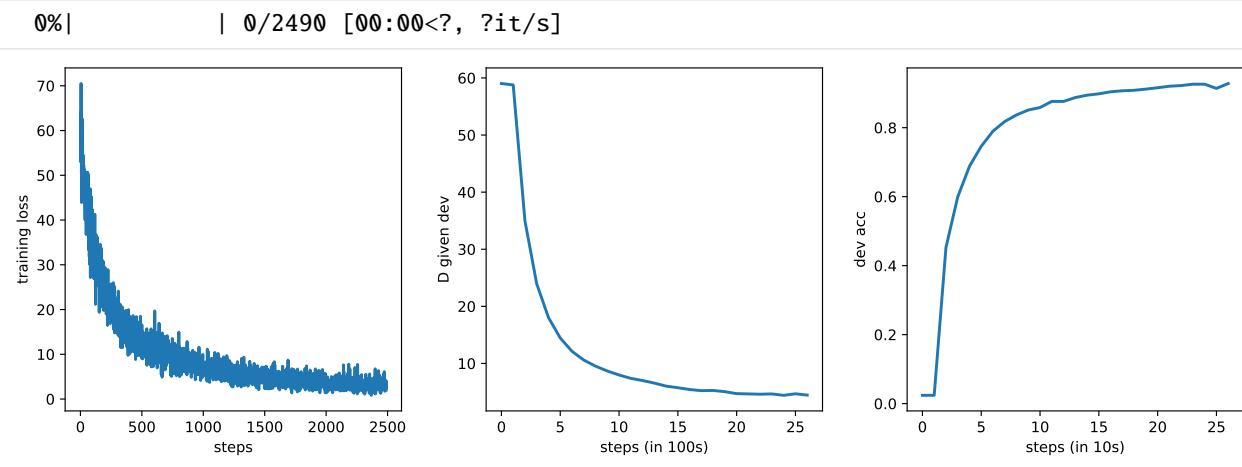
(continues on next page)

(continued from previous page)

```

)
(tag_embed): Embedding(16, 4)
(decoder): LSTM(4, 8, batch_first=True)
(logits_predictor): Sequential(
    (0): Dropout(p=0.0, inplace=False)
    (1): Linear(in_features=24, out_features=8, bias=True)
    (2): ReLU()
    (3): Dropout(p=0.0, inplace=False)
    (4): Linear(in_features=8, out_features=16, bias=True)
)
(init_state): Sequential(
    (0): Dropout(p=0.0, inplace=False)
    (1): Linear(in_features=16, out_features=8, bias=True)
    (2): Tanh()
)
(init_cell): Sequential(
    (0): Dropout(p=0.0, inplace=False)
    (1): Linear(in_features=16, out_features=8, bias=True)
    (2): Tanh()
)
)
Model size: 15,456 parameters

```



```

[58]: tagger_dev_data = TaggedCorpus(tagger_dev_x, tagger_dev_y, word_vocab, tag_vocab)
tagger_dev_dl = DataLoader(tagger_dev_data, batch_size=1, shuffle=False, collate_
                           fn=tagger_dev_data.pad_to_longest)
for batch_x, batch_y in tagger_dev_dl:
    print("Sample")
    y_out = ar_tagger.sample(batch_x.to(my_device))

    for x, y in zip(word_vocab.batch_decode(batch_x, strip_pad=True), tag_vocab.batch_
                     decode(y_out, strip_pad=False)):
        print(" ".join(f'{w}/{t}' for w, t in zip(x, y)))

    print("\nGreedy")

    y_out = ar_tagger.greedy(batch_x.to(my_device))

```

(continues on next page)

(continued from previous page)

```

for x, y in zip(word_vocab.batch_decode(batch_x, strip_pad=True), tag_vocab.batch_
    decode(y_out, strip_pad=False)):
    print(" ".join(f"{w}/{t}" for w, t in zip(x, y)))

break

```

Sample

-UNK-/NOUN ,/. a/DET -UNK-/NOUN -UNK-/VERB the/DET -UNK-/NOUN of/ADP -UNK-/ADJ -UNK-/
 -UNK-/NOUN ,/. is/VERB marketed/VERB *-1/X as/ADP a/DET \$/. -UNK-/NUM *u*/X -UNK-/VERB for/
 -ADP -UNK-/ADJ -UNK-/NOUN ./. -EOS/-EOS-

Greedy

-UNK-/NOUN ,/. a/DET -UNK-/NOUN -UNK-/VERB the/DET -UNK-/NOUN of/ADP -UNK-/ADJ -UNK-/
 -UNK-/NOUN ,/. is/VERB marketed/VERB *-1/X as/ADP a/DET \$/. -UNK-/NUM *u*/X -UNK-/VERB for/
 -ADP -UNK-/NOUN -UNK-/NOUN ./. -EOS/-EOS-

4.32.9 8. Exercise: design your own deep probabilistic model (optional, non-graded)

Now it's time to design your own probabilistic model. The objective is to find a dataset and design a probabilistic model for it, parameterised by a neural network. Be conscious about the distributions you choose to model your data with, make sure they are appropriate for the data! Use maximum likelihood estimation for estimating neural network parameters. Also, be conscious about the decision rule you use. Can you obtain the mode exactly and is it appropriate under your model? Is there perhaps another criterion more sensible? Can you think of a way to use maximisation of expected utility for your model? As we have already provided many helper classes for learning features from text data, we encourage you to look into textual datasets, but note the data you are modelling can still be of any domain: ordinal, real numbers, discrete, structured, etc. Some good places to look for such data are:

- NLTK: https://www.nltk.org/nltk_data/
- HuggingFace: <https://huggingface.co/datasets>
- Kaggle: <https://www.kaggle.com/datasets>

Remarks on other types of data:

- some structured data are fixed-dimensional (e.g., an image of size HxWxC), some models are built upon a chain rule factorisation just like the sequence labeller above, but their parameterisation might exploit the fact that the dimensionality is fixed (a good example is a MADE, you can ask Wilker more about those);
- some structured data are made of continuous parts (e.g., a time series in the finance domain, or in medical applications), in those cases your conditional factors are from a numerical family (e.g., Poisson, Normal, Gamma, etc), sometimes you need a distribution more powerful than the typical unimodal exponential family, in those cases you can use a mixture Model (we discuss those in the second module in the series) or a normalising flow (you can ask Wilker more about those, but note that Stratis will discuss NFs along with other advanced generative models)

4.33 DPM2 - Variational inference for deep discrete latent variable models

Filled notebook:

Recordings: Lecture 1.3, Lecture 1.4 & Lecture 1.5

Authors: Wilker Aziz

4.33.1 0. Intended Learning Outcomes

After this tutorial the student should be able to

- parameterise a latent variable model with discrete latent variables
- estimate parameters using exact log-likelihood function (when tractable)
- estimate parameters using neural variational inference

0.1 Setting up

```
[1]: import torch
import numpy as np
import random
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
import torch.distributions as td

from itertools import chain
from collections import defaultdict, OrderedDict
from tqdm.auto import tqdm

from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For having SVG graphics
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0

%matplotlib inline
```

```
[2]: def seed_all(seed=42):
    np.random.seed(seed)
    random.seed(seed)
    torch.manual_seed(seed)

seed_all()
```

4.33.2 1. Data

We are going to use toy image datasets for this notebook. These are fixed-dimensional observations for which encoder and decoders are relatively easy to design. This way we can focus on the aspects that are probabilistic in nature.

```
[3]: from torchvision.datasets import FashionMNIST
from torchvision import transforms
from torchvision.transforms import ToTensor
from torch.utils.data import random_split, Dataset
from torch.utils.data.dataloader import DataLoader
from torchvision.utils import make_grid
import torch.optim as opt

DATASET_PATH = "../data"
my_device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
```

A helper to binarize datasets:

```
[4]: class Binarizer(Dataset):

    def __init__(self, ds, threshold=0.5):
        self._ds = ds
        self._threshold = threshold

    def __len__(self):
        """Size of the corpus in number of sequence pairs"""
        return len(self._ds)

    def __getitem__(self, idx):
        """
        Return corpus_x[idx] and corpus_y[idx] converted to codes
        the latter has the EOS code in the end
        """
        x, y = self._ds[idx]
        return (x >= self._threshold).float(), y
```

FashionMNIST

```
[5]: dataset = FashionMNIST(root=DATASET_PATH, train=True, download=True,
                           transform=transforms.Compose([transforms.Resize(64), transforms.
                           ToTensor()]))
```

```
[6]: img_shape = dataset[0][0].shape
print("Shape of an image:", img_shape)
Shape of an image: torch.Size([1, 64, 64])
```

Let's make a dev set for ourselves:

```
[7]: val_size = 1000
train_size = len(dataset) - val_size
train_ds, val_ds = random_split(dataset, [train_size, val_size])
len(train_ds), len(val_ds)
```

[7]: (59000, 1000)

we suggest that you binarize the data in a first pass through this notebook, but as you will see, we can also model the continuous pixel intensities.

[8]: bin_data = True

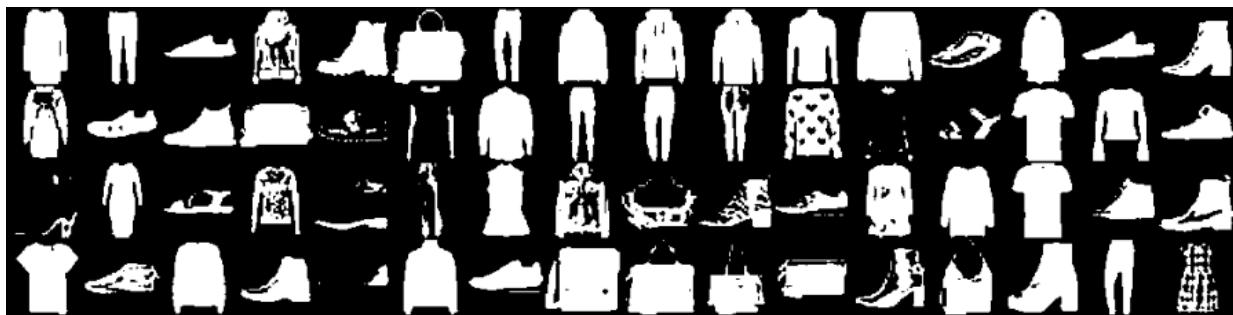
```
[9]: if bin_data:
    train_ds = Binarizer(train_ds)
    val_ds = Binarizer(val_ds)
```

```
[10]: batch_size = 64
train_loader = DataLoader(train_ds, batch_size, shuffle=True, num_workers=2, pin_
    _memory=True)
val_loader = DataLoader(val_ds, batch_size, num_workers=2, pin_memory=True)
```

Let's visualise a few samples

```
[11]: for images, y in train_loader:
    print('images.shape:', images.shape)
    plt.figure(figsize=(16,8))
    plt.axis('off')
    plt.imshow(make_grid(images, nrow=16).permute((1, 2, 0)))
    plt.show()
    break

images.shape: torch.Size([64, 1, 64, 64])
```



4.33.3 2. Latent variable models

We will be using NNs to parameterise a latent variable model, that is, a joint distribution over a collection of random variables (rvs), some of which are observed, some of which are not.

We are interested in two random variables (rvs):

- a discrete latent code $Z \in \mathcal{Z}$
- and an image $X \in \mathcal{X} \subseteq \mathbb{R}^D$

In this tutorial, x has a number C of channels, a certain width W and a certain height H , so $\mathcal{X} \subseteq \mathbb{R}^{C \times W \times H}$. Because we have fixed $D = C \times W \times H$, \mathcal{X} is finite-dimensional, but this need not be the case in general (for example, in a different domain, \mathcal{X} could be the unbounded space of all sentences of arbitrary length). We may treat the pixel intensities as discrete or continuous, as long as we choose an appropriate pmf/pdf for each case.

In this tutorial we will look into two types of latent code. A categorical code $z \in \{1, \dots, K\}$, and a combinatorial code $z \in \{0, 1\}^K$, in both cases \mathcal{Z} is countably finite, but in general this need not be the case (for example, we could have $z \in \mathbb{N}$ or z be a latent sequence of arbitrary length).

In this tutorial, we specify a joint distribution over $\mathcal{Z} \times \mathcal{X}$ by specifying a joint probability density function (pdf):

$$p_{ZX}(z, x|\theta) = p_Z(z|\theta)p_{X|Z}(x|z, \theta) \quad (4.55)$$

Here θ denotes the parameters of the NNs that parameterise the pmf p_Z and the pdf $p_{X|Z=z}$ (for any given z).

In this tutorial, the prior is fixed, but in general it need not be. We do not have additional predictors to condition on, but in some application domains you may have (e.g., in imagine captioning, we may be interested in a joint for a caption y and a latent code z given an image x ; in image generation, we may be interested in a joint distribution for an image x and a latent code z given a caption y).

2.1 Prior networks

We begin by specifying the component that parameterises the prior p_Z .

A prior network is an NN that parameterises a fixed prior distribution for the instances in a batch.

```
[12]: class PriorNet(nn.Module):
    """
    An NN that parameterises a prior distribution.

    For this lab, our priors are fixed, so this NN's forward pass
    simply returns a fixed prior with a given batch_shape.
    """

    def __init__(self, outcome_shape: tuple):
        """
        outcome_shape: this is the shape of a single outcome
        if you use a single integer k, we will turn it into (k,)
        """
        super().__init__()
        if isinstance(outcome_shape, int):
            outcome_shape = (outcome_shape,)
        self.outcome_shape = outcome_shape

    def forward(self, batch_shape):
        """
        Returns a td object for the batch.
        """
        raise NotImplementedError("Implement me!")

```

Let's implement two priors.

A product of (uniform) Bernoulli distributions

Here the latent code is a K -dimensional bit vector, you can think of each coordinate of the code as an attribute of the data point. We use a uniform prior per coordinate:

$$p_Z(z) = \prod_{k=1}^K \text{Bernoulli}(z_k|0.5) \quad (4.56)$$

Uniform (one-hot) Categorical distribution

Here the latent code is the a class from a discrete set $\{1, \dots, K\}$, we use a uniform prior over classes:

$$p_Z(z) = \text{Categorical}(z|K^{-1}\mathbf{1}_K) \quad (4.57)$$

where $\mathbf{1}_K$ is the K -dimensional vector of ones.

In practice, we can use the “OneHotCategorical” distribution, which wraps Categorical samples around a call to `onehot(sample, K)`.

```
[13]: class BernoulliPriorNet(PriorNet):
    """
    For z a D-dimensional bit vector:
    p(z) = prod_d Bernoulli(z[d]/0.5)
    """

    def __init__(self, outcome_shape):
        super().__init__(outcome_shape)
        # the product of Bernoulli priors will have Bernoulli(0.5) factors
        self.register_buffer("logits", torch.zeros(self.outcome_shape, requires_
        grad=False).detach())

    def forward(self, batch_shape):
        shape = batch_shape + self.outcome_shape
        # we wrap around td.Independent to obtain a pmf over multivariate draws
        # without td.Independent, we would have multiple pmfs, rather than one
        # pmf over a multivariate outcome space
        # td.Independent will interpret the rightmost dimensions as part of
        # the shape of the outcome
        return td.Independent(td.Bernoulli(logits=self.logits.expand(shape)), len(self.
        outcome_shape))

class CategoricalPriorNet(PriorNet):
    """
    For z the one-hot encoding of a category in a set of K categories:
    p(z) = OneHotCategorical(z/torch.ones(K) / K)
    """

    def __init__(self, outcome_shape):
        super().__init__(outcome_shape)
        self.register_buffer("logits", torch.zeros(self.outcome_shape, requires_
        grad=False).detach())

    def forward(self, batch_shape):
        shape = batch_shape + self.outcome_shape
        # OneHotCategorical is a wrapper around Categorical,
        # after drawing a Categorical sample, td.OneHotCategorical
        # encodes it using onehot(sample, support_size)
        # Here we do not need td.Independent, because OneHotCategorical
        # is a distribution over a multivariate draw (the one-hot
        # encoding of a category), which is different from the product of
        # Bernoulli prior
        return td.OneHotCategorical(logits=self.logits.expand(shape))
```

```
[14]: def test_priors(batch_size=3):
    prior_net = BernoulliPriorNet(7)
    print("\nBernoulli")
    print(f" outcome_shape={prior_net.outcome_shape}")
    p = prior_net(batch_shape=(batch_size,))
    print(f" distribution: {p}")
    z = p.sample()
    print(f" sample: {z}")
    print(f" shapes: sample={z.shape} log_prob={p.log_prob(z).shape}")

    prior_net = CategoricalPriorNet(7)
    print("\nCategorical")
    print(f" outcome_shape={prior_net.outcome_shape}")
    p = prior_net(batch_shape=(batch_size,))
    print(f" distribution: {p}")
    z = p.sample()
    print(f" sample: {z}")
    print(f" shapes: sample={z.shape} log_prob={p.log_prob(z).shape}")


test_priors()
```

```
Bernoulli
outcome_shape=(7,)
distribution: Independent(Bernoulli(logits: torch.Size([3, 7])), 1)
sample: tensor([[0., 0., 0., 1., 1., 1., 0.],
               [1., 0., 1., 0., 1., 0., 0.],
               [1., 0., 0., 1., 1., 1., 1.]])
shapes: sample=torch.Size([3, 7]) log_prob=torch.Size([3])
```

```
Categorical
outcome_shape=(7,)
distribution: OneHotCategorical()
sample: tensor([[0., 0., 1., 0., 0., 0., 0.],
               [1., 0., 0., 0., 0., 0., 0.],
               [0., 0., 0., 0., 1., 0., 0.]])
shapes: sample=torch.Size([3, 7]) log_prob=torch.Size([3])
```

2.2 Conditional probability distributions

Next, we create code to parameterise conditional probability distributions (cpds), which we do by having an NN parameterise a choice of pmf/pdf. This will be useful in parameterising the $p_{X|Z=z}$ component of our latent variable models (and, later on, it will also be useful for variational inference, when we develop $q_{Z|X=x}$).

Our general strategy is to map from a number of inputs (which the user will choose) to the parameters of a pmf/pdf support by `torch.distributions`.

```
[15]: class CPDNet(nn.Module):
    """
    Let L be a choice of distribution
    and x ~ L is an outcome with shape outcome_shape
```

(continues on next page)

(continued from previous page)

This is an NN whose forward method maps from a number of inputs to the parameters of L's pmf/pdf and returns a torch.distributions object representing L's pmf/pdf.

"""

```
def __init__(self, outcome_shape):
    """
    outcome_shape: this is the shape of a single outcome
                   if you use a single integer k, we will turn it into (k,)
    """
    super().__init__()
    if isinstance(outcome_shape, int):
        outcome_shape = (outcome_shape,)
    self.outcome_shape = outcome_shape

def forward(self, inputs):
    """
    Return a torch.distribution object predicted from `inputs`.

    inputs: a tensor with shape batch_shape + (num_inputs,)
    """
    raise NotImplementedError("Implemented me")
```

2.2.1 Observational model

The observational model prescribes the distribution of $X|Z = z$.

If we assume our pixel intensities are binary, we can use a product of $C \times W \times H$ Bernoulli distributions, which we parameterise jointly using an NN:

$$p_{X|Z}(x|z, \theta) = \prod_{c=1}^C \prod_{w=1}^W \prod_{h=1}^H \text{Bernoulli}(x_{c,w,h}|f_{c,w,h}(z; \theta)) \quad (4.58)$$

Here $\mathbf{f}(z; \theta) \in (0, 1)^C \times (0, 1)^W \times (0, 1)^H$ is an NN architecture such as a feed-forward net or a stack of transposed convolution layers. In NN literature, such architectures are often called *decoders*.

If we assume our pixel intensities are real values in $[0, 1]$ (0 and 1 included), we need to parameterise a pdf. A good choice of pdf is the [ContinuousBernoulli distributions](#), which is a single-parameter distribution (much like the Bernoulli) whose support is the set $[0, 1]$.

Let's start by designing \mathbf{f} .

A very basic design uses a FFNN:

```
[16]: class ReshapeLast(nn.Module):
    """
    Helper layer to reshape the rightmost dimension of a tensor.

    This can be used as a component of nn.Sequential.
    """

    def __init__(self, shape: tuple):
```

(continues on next page)

(continued from previous page)

```

shape: desired rightmost shape
"""
super().__init__()
self._shape = shape

def forward(self, input):
    # reshapes the last dimension into self.shape
    return input.reshape(input.shape[:-1] + self._shape)

def build_ffnn_decoder(latent_size, num_channels, width=64, height=64, hidden_size=512, ↴
← p_drop=0.):
    """
    Map the latent code to a tensor with shape [num_channels, width, height]
    using a FFNN with 2 hidden layers.

    latent_size: size of latent code
    num_channels: number of channels in the output
    width: image shape
    height: image shape
    hidden_size: we first map from latent_size to hidden_size and
        then use feed forward NNs to map it to [num_channels, width, height]
    p_drop: dropout rate before linear layers
    """
    decoder = nn.Sequential(
        nn.Dropout(p_drop),
        nn.Linear(latent_size, hidden_size),
        nn.ReLU(),
        nn.Dropout(p_drop),
        nn.Linear(hidden_size, hidden_size),
        nn.ReLU(),
        nn.Dropout(p_drop),
        nn.Linear(hidden_size, num_channels * width * height),
        ReshapeLast((num_channels, width, height)),
    )
    return decoder

```

[17]: # mapping from 10-dimensional latent code
build_ffnn_decoder(latent_size=10, num_channels=1)(torch.zeros((5, 10))).shape
[17]: torch.Size([5, 1, 64, 64])

[18]: # we can also have a structured batch shape (e.g., [3, 5])
build_ffnn_decoder(latent_size=10, num_channels=1)(torch.zeros((3, 5, 10))).shape
[18]: torch.Size([3, 5, 1, 64, 64])

The downside is that the output layer is rather large.

An architecture with inductive biases that are more appropriate for our data type is a CNN, in particular, a transposed CNN. Here we design one such decoder:

[19]: class MySequential(nn.Sequential):

(continues on next page)

(continued from previous page)

.....

This is a version of nn.Sequential that works with structured batches (i.e., batches that have multiple dimensions) even when some of the nn layers in it does not.

The idea is to just wrap nn.Sequential around two calls to reshape which remove and restore the batch dimensions.

.....

```

def __init__(self, *args, event_dims=1):
    super().__init__(*args)
    self._event_dims = event_dims

def forward(self, input):
    # memorise batch shape
    batch_shape = input.shape[:-self._event_dims]
    # memorise latent shape
    event_shape = input.shape[-self._event_dims:]
    # flatten batch shape and obtain outputs
    output = super().forward(input.reshape((-1,) + event_shape))
    # restore batch shape
    return output.reshape(batch_shape + output.shape[1:])

def build_cnn_decoder(latent_size, num_channels, width=64, height=64, hidden_size=1024, ↴
    ↪ p_drop=0.):
    """
    Map the latent code to a tensor with shape [num_channels, width, height].
    """

    latent_size: size of latent code
    num_channels: number of channels in the output
    width: must be 64 (for now)
    height: must be 64 (for now)
    hidden_size: we first map from latent_size to hidden_size and
        then use transposed 2d convolutions to [num_channels, width, height]
    p_drop: dropout rate before linear layers
    """
    if width != 64:
        raise ValueError("The width is hardcoded")
    if height != 64:
        raise ValueError("The height is hardcoded")

    # TODO: change the architecture so width and height are not hardcoded
    decoder = MySequential(
        nn.Dropout(p_drop),
        nn.Linear(latent_size, hidden_size),
        ReshapeLast((hidden_size, 1, 1)),
        nn.ConvTranspose2d(hidden_size, 128, 5, 2),
        nn.ReLU(),
        nn.ConvTranspose2d(128, 64, 5, 2),
        nn.ReLU(),
        nn.ConvTranspose2d(64, 32, 6, 2),

```

(continues on next page)

(continued from previous page)

```

        nn.ReLU(),
        nn.ConvTranspose2d(32, num_channels, 6, 2),
        event_dims=1
    )
    return decoder

```

[20]: # a batch of five 10-dimensional latent codes is transformed
into a batch of 5 images, each with shape [1,64,64]
build_cnn_decoder(latent_size=10, num_channels=1)(torch.zeros((5, 10))).shape
[20]: torch.Size([5, 1, 64, 64])

[21]: # note that because we use MySequential,
we can have a batch of [3, 5] assignments
(this is useful, for example, when we have multiple draws of the latent
variable for each of the data points in the batch)
build_cnn_decoder(latent_size=10, num_channels=1)(torch.zeros((3, 5, 10))).shape
[21]: torch.Size([3, 5, 1, 64, 64])

Now we are in position to design a CPDNet for our image model, it simply combines a choice of decoder with a choice of distribution:

[22]: **class BinarizedImageModel(CPDNet):**

```

    def __init__(self, num_channels, width, height, latent_size, decoder_type=build_ffnn_
decoder, p_drop=0.):
        super().__init__((num_channels, width, height))
        self.decoder = decoder_type(
            latent_size=latent_size,
            num_channels=num_channels,
            width=width,
            height=height,
            p_drop=p_drop
        )

    def forward(self, z):
        """
        Return the cpd X/Z=z

        z: batch_shape + (latent_dim,)
        """
        h = self.decoder(z)
        return td.Independent(td.Bernoulli(logits=h), len(self.outcome_shape))

class ContinuousImageModel(CPDNet):

    def __init__(self, num_channels, width, height, latent_size, decoder_type=build_ffnn_
decoder, p_drop=0.):
        super().__init__((num_channels, width, height))
        self.decoder = decoder_type(

```

(continues on next page)

(continued from previous page)

```

        latent_size=latent_size,
        num_channels=num_channels,
        width=width,
        height=height,
        p_drop=p_drop
    )

def forward(self, z):
    """
    Return the cpd X/Z=z

    z: batch_shape + (latent_dim,)
    """
    # batch_shape + (num_channels, width, height)
    h = self.decoder(z)
    return td.Independent(td.ContinuousBernoulli(logits=h), len(self.outcome_shape))

```

[23]:

```

obs_model = BinarizedImageModel(
    num_channels=img_shape[0],
    width=img_shape[1],
    height=img_shape[2],
    latent_size=10,
    p_drop=0.1,
)
print(obs_model)
# a batch of five zs is mapped to 5 distributions over [1,64,64]-dimensional
# binary tensors
print(obs_model(torch.zeros([5, 10])))

```

BinarizedImageModel(
 (decoder): Sequential(
 (0): Dropout(p=0.1, inplace=False)
 (1): Linear(in_features=10, out_features=512, bias=True)
 (2): ReLU()
 (3): Dropout(p=0.1, inplace=False)
 (4): Linear(in_features=512, out_features=512, bias=True)
 (5): ReLU()
 (6): Dropout(p=0.1, inplace=False)
 (7): Linear(in_features=512, out_features=4096, bias=True)
 (8): ReshapeLast()
)
)
Independent(Bernoulli(logits: torch.Size([5, 1, 64, 64])), 3)

We can also use a different decoder

[24]:

```

obs_model = BinarizedImageModel(
    num_channels=img_shape[0],
    width=img_shape[1],
    height=img_shape[2],
    latent_size=10,
    p_drop=0.1,
)

```

(continues on next page)

(continued from previous page)

```

    decoder_type=build_cnn_decoder
)
print(obs_model)
# a batch of five zs is mapped to 5 distributions over [1,64,64]-dimensional
# binary tensors
print(obs_model(torch.zeros([5, 10])))
BinarizedImageModel1(
    decoder): MySequential(
        (0): Dropout(p=0.1, inplace=False)
        (1): Linear(in_features=10, out_features=1024, bias=True)
        (2): ReshapeLast()
        (3): ConvTranspose2d(1024, 128, kernel_size=(5, 5), stride=(2, 2))
        (4): ReLU()
        (5): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2))
        (6): ReLU()
        (7): ConvTranspose2d(64, 32, kernel_size=(6, 6), stride=(2, 2))
        (8): ReLU()
        (9): ConvTranspose2d(32, 1, kernel_size=(6, 6), stride=(2, 2))
    )
)
Independent(Bernoulli(logits: torch.Size([5, 1, 64, 64])), 3)

```

2.3 Joint distribution

We can now combine a prior and an observational model into a joint distribution. A joint distribution supports a few important operations such as marginal and posterior pdf assessments, as well as sampling from the joint distribution. Marginal and posterior assessments require computations that may or may not be tractable, see below.

From a joint pdf, we can compute the marginal density of x via

$$p_X(x|\theta) = \sum_{z \in \mathcal{Z}} p_{ZX}(z, x|\theta) \quad (4.59)$$

$$= \sum_{z \in \mathcal{Z}} p_Z(z|\theta) p_{X|Z}(x|z, \theta) \quad (4.60)$$

Under the assumption that \mathcal{Z} is countably finite, this can be done via enumeration in time proportional to $\mathcal{O}(|\mathcal{Z}|)$. This marginalisation is therefore intractable for countably infinite \mathcal{Z} or for \mathcal{Z} that is combinatorially large (e.g., the space of K -dimensional bit vectors).

Assuming enumeration is tractable, we can also assess the posterior mass of z given x via

$$p_{Z|X}(z|x, \theta) = \frac{p_Z(z|\theta) p_{X|Z}(x|z, \theta)}{p_X(x|\theta)} \quad (4.61)$$

When marginalisation is intractable, we can obtain a naive lowerbound by direct application of Jensen's inequality:

$$\log p_X(x|\theta) = \log \sum_{z \in \mathcal{Z}} p_Z(z|\theta) p_{X|Z}(x|z, \theta) \quad (4.62)$$

$$\stackrel{\text{JL}}{\geq} \sum_{z \in \mathcal{Z}} p_Z(z|\theta) \log p_{X|Z}(x|z, \theta) \quad (4.63)$$

$$\stackrel{\text{MC}}{\approx} \frac{1}{S} \sum_{s=1}^S \log p_{X|Z}(x|z_s, \theta) \quad \text{where } z_s \sim p_Z \quad (4.64)$$

A better lowerbound could be obtained via importance sampling, but it would require training an approximating distribution (as we will do in variational inference).

Recall that, given a dataset \mathcal{D} , the log-likelihood function $\mathcal{L}(\theta|\mathcal{D}) = \sum_{x \in \mathcal{D}} \log p_X(x|\theta)$ requires performing marginal density assessments. Whenever exact marginalisation is intractable, we are unable to assess $\mathcal{L}(\theta|\mathcal{D})$ and its gradient with respect to θ . If the prior is fixed, we can use the naive lowerbound to obtain a gradient estimate, but, again, our naive application of JI leads to a generally rather loose bound.

```
[25]: class JointDistribution(nn.Module):
    """
    A wrapper to combine a prior net and a cpd net into a joint distribution.
    """

    def __init__(self, prior_net: PriorNet, cpd_net: CPDNet):
        """
        prior_net: object to parameterise p_Z
        cpd_net: object to parameterise p_{X|Z=z}
        """
        super().__init__()
        self.prior_net = prior_net
        self.cpd_net = cpd_net

    def prior(self, shape):
        return self.prior_net(shape)

    def obs_model(self, z):
        return self.cpd_net(z)

    def sample(self, shape):
        """
        Return z via prior_net(shape).sample()
        and x via cpd_net(z).sample()
        """
        pz = self.prior_net(shape)
        z = pz.sample()
        px_z = self.cpd_net(z)
        x = px_z.sample()
        return z, x

    def log_prob(self, z, x):
        """
        Assess the log density of the joint outcome.
        """
        batch_shape = z.shape[:-len(self.prior_net.outcome_shape)]
        pz = self.prior_net(batch_shape)
        px_z = self.cpd_net(z)
        return pz.log_prob(z) + px_z.log_prob(x)

    def log_marginal(self, x, enumerate_fn):
        """
        Return log marginal density of x.

        enumerate_fn: function that enumerates the support of the prior
        (this is needed for marginalisation  $p(x) = \int p(z, x) dz$ )
        """

```

(continues on next page)

(continued from previous page)

This only really makes sense if the support is a (small) countably finite set. In such cases, you can use enumerate=lambda p: p.enumerate_support() which is supported, for example, by Categorical and OneHotCategorical.

If the support is discrete (eg, bit vectors) you can still dare to enumerate it explicitly, but you will need to write customised code, as torch.distributions will not offer that functionality for you.

If the support is uncountable, countably infinite, or just large anyway, you need approximate tools (such as VI, importance sampling, etc)

```
"""
batch_shape = x.shape[:-len(self.cpd_net.outcome_shape)]
pz = self.prior_net(batch_shape)
log_joint = []
# (support_size,) + batch_shape
z = enumerate_fn(pz)
px_z = self.cpd_net(z)
# (support_size,) + batch_shape
log_joint = pz.log_prob(z) + px_z.log_prob(x.unsqueeze(0))
# batch_shape
return torch.logsumexp(log_joint, 0)
```

def posterior(self, x, enumerate_fn):

```
"""
Return the posterior distribution Z|X=x.
```

As the code is discrete, we return a discrete distribution over the complete space of all possible latent codes. This is done via exhaustive enumeration provided by `enumerate_fn`.

```
"""
batch_shape = x.shape[:-len(self.cpd_net.outcome_shape)]
pz = self.prior_net(batch_shape)
# (support_size,) + batch_shape
z = enumerate_fn(pz)
px_z = self.cpd_net(z)
# (support_size,) + batch_shape
log_joint = pz.log_prob(z) + px_z.log_prob(x.unsqueeze(0))
# batch_shape + (support_size,)
log_joint = torch.swapaxes(log_joint, 0, -1)
return td.Categorical(logits=log_joint)
```

def naive_lowerbound(self, x, num_samples: int):

```
"""
Return an MC lowerbound on log marginal density of x:
    log p(x) >= 1/S \sum_s log p(x|z[s])
        with z[s] ~ p_Z
"""

batch_shape = x.shape[:-len(self.cpd_net.outcome_shape)]
```

```
pz = self.prior_net(batch_shape)
# (num_samples,) + batch_shape + prior_outcome_shape
```

(continues on next page)

(continued from previous page)

```

log_probs = []
# I'm using a for loop, but note that with enough GPU memory
# one could parallelise this step
for z in pz.sample((num_samples,)):
    px_z = self.cpd_net(z)
    log_probs.append(px_z.log_prob(x))
# (num_samples,) + batch_shape
log_probs = torch.stack(log_probs)
# batch_shape
return torch.mean(log_probs, 0)

```

```

[26]: def test_joint_dist(latent_size=10, data_shape=(1, 64, 64), batch_size=2, hidden_
      size=32):
    p = JointDistribution(
        prior_net=CategoricalPriorNet(latent_size),
        cpd_net=BinarizedImageModel(
            num_channels=data_shape[0],
            width=data_shape[1],
            height=data_shape[2],
            latent_size=latent_size,
            decoder_type=build_cnn_decoder
        )
    )
    print("Model for binarized data")
    print(p)
    z, x = p.sample((batch_size,))
    print("sampled z")
    print(z)
    print("sampled x")
    print(x)
    enumerate_fn = lambda p: p.enumerate_support()
    print("marginal")
    print(p.log_marginal(x, enumerate_fn))
    print("MC lowerbound")
    print(" 1:", p.naive_lowerbound(x, 10))
    print(" 2:", p.naive_lowerbound(x, 10))
    print("posterior distribution")
    print(p.posterior(x, enumerate_fn).probs)

    print("\n\n")

    p = JointDistribution(
        prior_net=CategoricalPriorNet(latent_size),
        cpd_net=ContinuousImageModel(
            num_channels=data_shape[0],
            width=data_shape[1],
            height=data_shape[2],
            latent_size=latent_size,
            decoder_type=build_cnn_decoder
        )
    )
    print("Model for continuous data")

```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```

[0., 0., 0., ..., 1., 1., 1.],
[1., 0., 0., ..., 0., 0., 0.],
[1., 0., 1., ..., 1., 0., 0.]]])
marginal
tensor([-2833.0503, -2840.0588], grad_fn=<LogsumexpBackward0>)
MC lowerbound
1: tensor([-2833.0542, -2840.0547], grad_fn=<MeanBackward1>)
2: tensor([-2833.0972, -2840.0427], grad_fn=<MeanBackward1>)
posterior distribution
tensor([[0.0848, 0.1013, 0.1000, 0.1056, 0.0816, 0.0984, 0.1063, 0.1045, 0.1015,
       0.1161],
       [0.0965, 0.1006, 0.0892, 0.0886, 0.1072, 0.1042, 0.1156, 0.1104, 0.1010,
       0.0867]], grad_fn=<SoftmaxBackward0>)

Model for continuous data
JointDistribution(
    (prior_net): CategoricalPriorNet()
    (cpd_net): ContinuousImageModel(
        (decoder): MySequential(
            (0): Dropout(p=0.0, inplace=False)
            (1): Linear(in_features=10, out_features=1024, bias=True)
            (2): ReshapeLast()
            (3): ConvTranspose2d(1024, 128, kernel_size=(5, 5), stride=(2, 2))
            (4): ReLU()
            (5): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2))
            (6): ReLU()
            (7): ConvTranspose2d(64, 32, kernel_size=(6, 6), stride=(2, 2))
            (8): ReLU()
            (9): ConvTranspose2d(32, 1, kernel_size=(6, 6), stride=(2, 2))
        )
    )
)
sampled z
tensor([[0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]])
sampled x
tensor([[[[0.0177, 0.4419, 0.8944, ..., 0.1530, 0.0298, 0.2958],
          [0.3006, 0.2818, 0.9756, ..., 0.3906, 0.3666, 0.5736],
          [0.5353, 0.1523, 0.1889, ..., 0.1321, 0.3172, 0.1197],
          ...,
          [0.5384, 0.4939, 0.3950, ..., 0.1797, 0.4078, 0.8856],
          [0.8161, 0.1738, 0.2747, ..., 0.3477, 0.0262, 0.0069],
          [0.5498, 0.7573, 0.8136, ..., 0.1438, 0.7143, 0.1651]],
         [[[0.3061, 0.7684, 0.9373, ..., 0.3133, 0.9556, 0.9470],
           [0.6662, 0.7195, 0.2499, ..., 0.1698, 0.7220, 0.4024],
           [0.9018, 0.6794, 0.7495, ..., 0.5036, 0.7543, 0.2213],
           ...,
           [0.9679, 0.9352, 0.9314, ..., 0.1054, 0.8021, 0.6079],

```

(continues on next page)

(continued from previous page)

```
[0.4803, 0.2586, 0.0312, ..., 0.7962, 0.8062, 0.9996],
[0.2138, 0.5723, 0.5005, ..., 0.7090, 0.4256, 0.8469]]])
marginal
tensor([2.4405, 1.3252], grad_fn=<LogsumexpBackward0>)
MC lowerbound
1: tensor([2.4240, 1.3365], grad_fn=<MeanBackward1>)
2: tensor([2.4259, 1.3594], grad_fn=<MeanBackward1>)
posterior distribution
tensor([[0.1080, 0.1020, 0.0954, 0.1017, 0.0990, 0.1036, 0.0978, 0.0957, 0.0963,
       0.1006],
       [0.0995, 0.1146, 0.0970, 0.1074, 0.1070, 0.0973, 0.0960, 0.0972, 0.0957,
       0.0882]], grad_fn=<SoftmaxBackward0>)
```

4.33.4 3. Learning

We estimate θ using stochastic gradient-based maximum likelihood estimation. For a tractable model, we can assess the log-likelihood function

$$\mathcal{L}(\theta|\mathcal{D}) = \sum_{x \in \mathcal{D}} \log p_X(x|\theta) \quad (4.65)$$

and estimate $\nabla_\theta \mathcal{L}(\theta|\mathcal{D})$ using random mini-batches:

$$\nabla_\theta \mathcal{L}(\theta|\mathcal{D}) \stackrel{\text{MC}}{\approx} \frac{1}{S} \sum_{s=1}^S \nabla_\theta \log p_X(x^{(s)}|\theta) \quad (4.66)$$

$$\text{where } x^{(s)} \sim \mathcal{D} \quad (4.67)$$

3.1 Tractable LVMs

A mixture model with K components is in principle a tractable LVM:

$$p_X(x|\theta) = \sum_{z=1}^K \frac{1}{K} p_{X|Z}(x|z, \theta) \quad (4.68)$$

For $p_{X|Z=z}$ we use the image model above, which conditions on the one-hot encoding of z .

```
[27]: print(JointDistribution(
    prior_net=CategoricalPriorNet(10),
    cpd_net=BinarizedImageModel(
        num_channels=img_shape[0],
        width=img_shape[1],
        height=img_shape[2],
        latent_size=10,
        decoder_type=build_ffnn_decoder
    )
))
JointDistribution(
    (prior_net): CategoricalPriorNet()
    (cpd_net): BinarizedImageModel()
```

(continues on next page)

(continued from previous page)

```
(decoder): Sequential(
    (0): Dropout(p=0.0, inplace=False)
    (1): Linear(in_features=10, out_features=512, bias=True)
    (2): ReLU()
    (3): Dropout(p=0.0, inplace=False)
    (4): Linear(in_features=512, out_features=512, bias=True)
    (5): ReLU()
    (6): Dropout(p=0.0, inplace=False)
    (7): Linear(in_features=512, out_features=4096, bias=True)
    (8): ReshapeLast()
)
)
)
```

3.2 Training algorithm

Here we have some helper code to assess and train an LVM using its exact log-likelihood function.

```
[28]: def assess_exact(model: JointDistribution, enumerate_fn, dl, device):
    """
    Wrapper for estimating the model's log-likelihood.
    As data samples, we use all data points in a data loader.

    model: a joint distribution for which Z can be exactly marginalised
    enumerate_fn: algorithm to enumerate the support of Z for a batch
        this will be used to assess `model.log_prob(batch, enumerate_fn)`
    dl: torch data loader
    device: torch device
    """
    L = 0
    data_size = 0
    with torch.no_grad():
        for batch_x, batch_y in dl:
            L = L + model.log_marginal(batch_x.to(device), enumerate_fn).sum(0)
            data_size += batch_x.shape[0]
    L = L / data_size
    return L
```

```
[29]: def train_exact(model: JointDistribution, enumerate_fn, optimiser,
    training_data, dev_data,
    batch_size=64, num_epochs=10, check_every=10,
    grad_clip=5.,
    report_metrics=['L'],
    num_workers=2,
    device=torch.device('cuda:0'))
):
    """
    model: a joint distribution where Z can be exactly marginalised
    enumerate_fn: function to enumerate the support of Z
        (used in combination with model.log_prob)
    optimiser: pytorch optimiser
```

(continues on next page)

(continued from previous page)

```

training_corpus: a torchvision corpus for training
dev_corpus: a torchvision corpus for validation
batch_size: use more if you have more memory
num_epochs: use more for improved convergence
check_every: use less to check performance on dev set more often
device: where we run the experiment

Return a log of quantities computed during training (for plotting)
"""

batcher = DataLoader(training_data, batch_size, shuffle=True, num_workers=num_
workers, pin_memory=True)
dev_batcher = DataLoader(dev_data, batch_size, num_workers=num_workers, pin_
memory=True)

total_steps = num_epochs * len(batcher)
log = defaultdict(list)

step = 0
model.eval()

log['dev.L'].append((step, assess_exact(model, enumerate_fn, dev_batcher, u
device=device).item()))

with tqdm(range(total_steps)) as bar:
    for epoch in range(num_epochs):
        for batch_x, batch_y in batcher:
            model.train()
            optimiser.zero_grad()

            loss = - model.log_marginal(
                batch_x.to(device),
                enumerate_fn
            ).mean(0)
            log['loss'].append((step, loss.item()))

            loss.backward()

            nn.utils.clip_grad_norm_(
                model.parameters(),
                grad_clip
            )
            optimiser.step()

            bar_dict = OrderedDict()
            bar_dict['loss'] = f'{loss.item():.2f}'
            bar_dict[f'dev.L'] = f'{:.2f}'.format(log[f'dev.L'][-1][1])
            bar.set_postfix(bar_dict)
            bar.update()

        if step % check_every == 0:

```

(continues on next page)

(continued from previous page)

```

        model.eval()
        log['dev.L'].append((step, assess_exact(model, enumerate_fn, dev_
batcher, device=device).item()))

    step += 1

    model.eval()
    log['dev.L'].append((step, assess_exact(model, enumerate_fn, dev_batcher,_
device=device).item()))

return log

```

Here's some helper code to inspect samples from a mixture model

```
[30]: def inspect_mixture_model(model: JointDistribution, support_size):
    # Display some samples from the prior
    _, x_ = model.sample((support_size, 5))
    x_ = x_.cpu().reshape(-1, 1, 64, 64)
    plt.figure(figsize=(12,6))
    plt.axis('off')
    plt.imshow(make_grid(x_, nrow=support_size).permute((1, 2, 0)))
    plt.title("Prior samples")
    plt.show()

    # Display samples from each of the 10 components
    # [support_size, latent_dim]
    support = model.prior(tuple()).enumerate_support()
    # [support_size, 1, 64, 64]
    x_ = model.obs_model(support).sample((5,)).cpu().reshape(-1, 1, 64, 64)
    plt.figure(figsize=(12,6))
    plt.axis('off')
    plt.imshow(make_grid(x_, nrow=support_size).permute((1, 2, 0)))
    plt.title("Component samples")
    plt.show()
```

3.2.1 Experiment

```
[31]: seed_all()

mixture_model = JointDistribution(
    prior_net=CategoricalPriorNet(10),
    cpd_net=BinarizedImageModel( # could also be continuous (but then change the_
    preprocessing in the beginning)
        num_channels=img_shape[0],
        width=img_shape[1],
        height=img_shape[2],
        latent_size=10,
        decoder_type=build_ffnn_decoder, # could also be CNN (but consider less latent_
        classes, as the transposed CNN decoder is a bit too slow for exact enumeration)
        p_drop=0.1,
```

(continues on next page)

(continued from previous page)

```
        )
    ).to(my_device)

print(mixture_model)

mm_optim = opt.Adam(mixture_model.parameters(), lr=1e-3, weight_decay=1e-6)

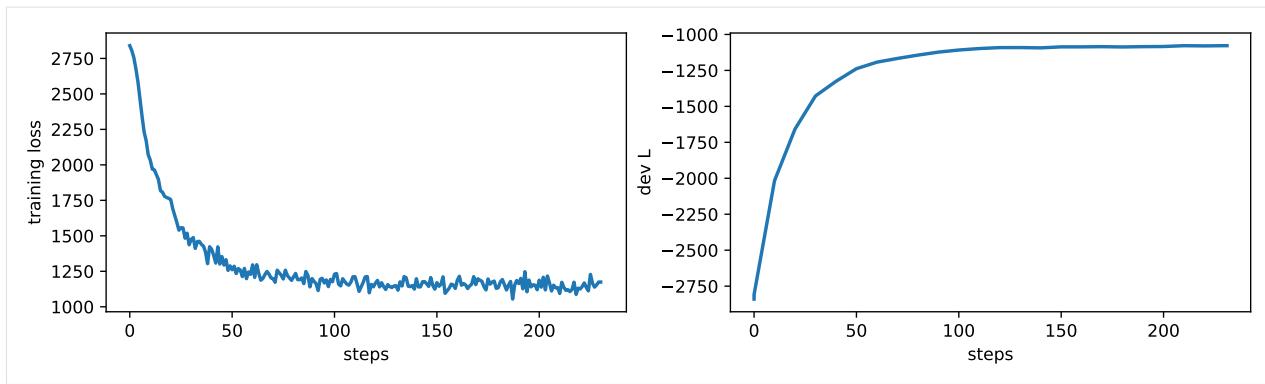
JointDistribution(
    (prior_net): CategoricalPriorNet()
    (cpd_net): BinarizedImageModel(
        (decoder): Sequential(
            (0): Dropout(p=0.1, inplace=False)
            (1): Linear(in_features=10, out_features=512, bias=True)
            (2): ReLU()
            (3): Dropout(p=0.1, inplace=False)
            (4): Linear(in_features=512, out_features=512, bias=True)
            (5): ReLU()
            (6): Dropout(p=0.1, inplace=False)
            (7): Linear(in_features=512, out_features=4096, bias=True)
            (8): ReshapeLast()
        )
    )
)
```

```
[32]: log = train_exact(  
    model=mixture_model,  
    optimiser=mm_optim,  
    enumerate_fn=lambda p: p.enumerate_support(),  
    training_data=train_ds,  
    dev_data=val_ds,  
    batch_size=256,  
    num_epochs=1, # use more for a better model  
    check_every=10,  
    grad_clip=5.,  
    device=my_device  
)
```

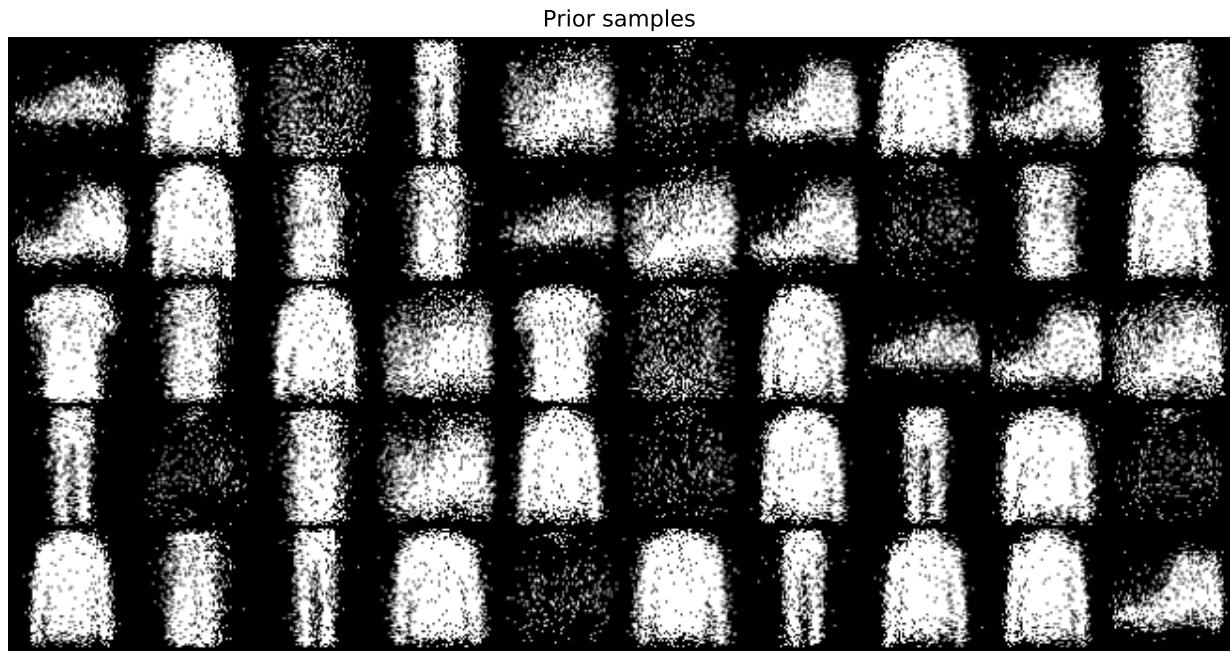
```
[33]: fig, axs = plt.subplots(1, 2, sharex=True, sharey=False, figsize=(12, 3))

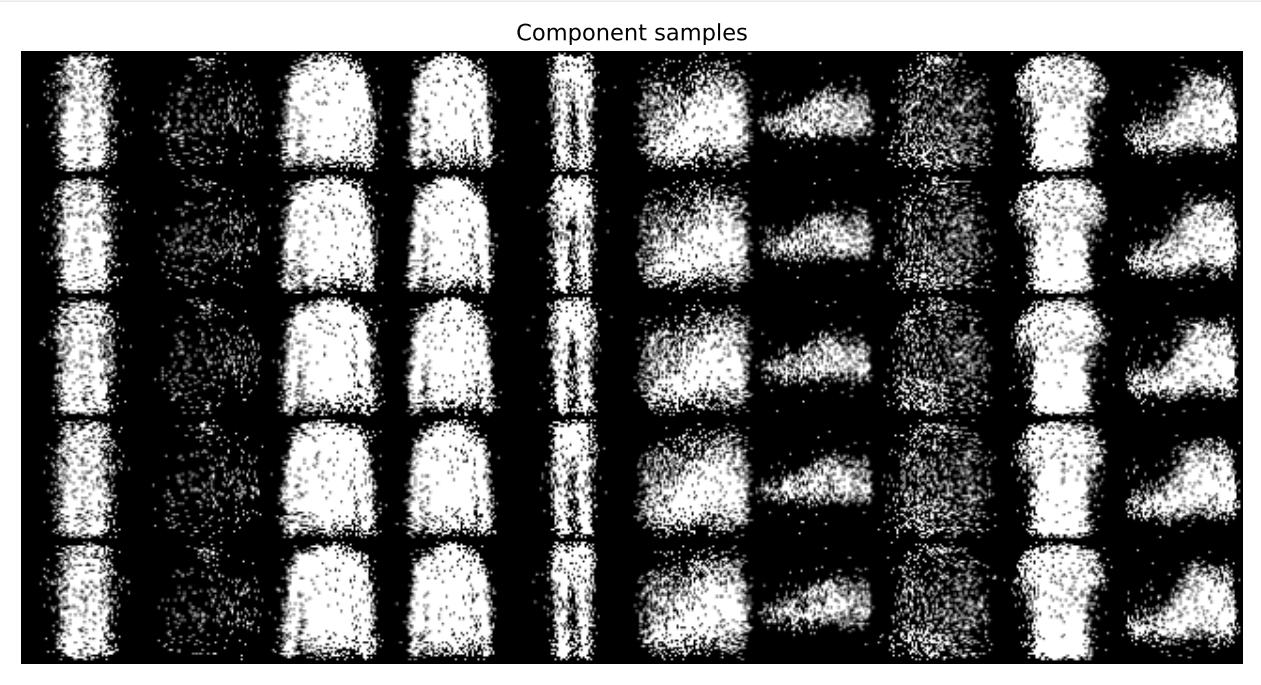
_ = axs[0].plot(np.array(log['loss'])[:,0], np.array(log['loss'])[:,1])
_ = axs[0].set_ylabel("training loss")
_ = axs[0].set_xlabel("steps")

_ = axs[1].plot(np.array(log['dev.L'])[:,0], np.array(log['dev.L'])[:,1])
_ = axs[1].set_ylabel("dev L")
_ = axs[1].set_xlabel("steps")
```



```
[34]: inspect_mixture_model(mixture_model, 10)
```





A mixture model offers very limited capacity to the observational model, it can at best partition the data space into K groups.

A factor model instead offers up 2^K codes to the observational model, next we will design a factor model.

3.3 Intractable LVMs

When marginalisation is intractable (e.g., $\mathcal{Z} = \{0, 1\}^K$ for moderate K) or just plain inconvenient (e.g., $\mathcal{Z} = \{1, \dots, K\}$ for large K , e.g., $K > 10$), we resort to variational inference introducing a parametric approximation $q_{Z|X=x}$ to the model's true posterior distribution $p_{Z|X=x}$ and estimate both the approximation and the joint distribution by maximising the evidence lowerbound (ELBO), show below for a single observation x :

$$\mathcal{E}(\lambda, \theta | \mathcal{D}) = \mathbb{E}_{x \sim \mathcal{D}} \left[\mathbb{E} \left[\log \frac{p_{ZX}(z, x | \theta)}{q_{Z|X}(z | x, \lambda)} \right] \right] \quad (4.69)$$

$$= \underbrace{\mathbb{E}_{x \sim \mathcal{D}} [\mathbb{E}[\log p_{ZX}(x | z, \theta)]]}_{-D} - \underbrace{\mathbb{E}_{x \sim \mathcal{D}} [\text{KL}(q_{Z|X=x} || p_Z)]}_{R} \quad (4.70)$$

where the inner expectation is taken with respect to $q_{Z|X}(z | x, \lambda)$.

When computed in expectation under the data distribution, the two components of the ELBO, namely, the expected log-likelihood $\mathbb{E}[\log p_{ZX}(x | z, \theta)]$ and the ‘‘KL term’’ $\text{KL}(q_{Z|X=x} || p_Z)$, are related to two information-theoretic quantities known as *distortion* and *rate*.

We choose our approximation to be such that: its support is embedded in the support of the prior, it is simple enough to sample from, and it is simple enough to assess the mass of a sample. If possible, we choose it such that other quantities are also tractable (e.g., entropy, relative entropy).

For a multivariate z , we normally choose a factorised family, for example, if z is a D -dimensional bit vector:

$$q_{Z|X}(z | x, \lambda) = \prod_{k=1}^K \text{Bernoulli}(z_k | g_k(x; \lambda)) \quad (4.71)$$

with $\mathbf{g}(x; \lambda) \in (0, 1)^K$. This is called a *mean field assumption*.

For a Categorical z , the marginalisation is tractable in principle (it takes time linear in the number K of possible assignments), but for certain decoders, even K forward passes might be too much, so we use an independently parameterised Categorical distribution:

$$q_{Z|X}(z|x, \lambda) = \text{Categorical}(z|\mathbf{g}(x; \lambda)) \quad (4.72)$$

with $\mathbf{g}(x; \lambda) \in \Delta_{K-1}$.

3.4 Inference model

The inference model is a conditional model of the latent variable, for which we design CPD nets.

Before we go on, it is useful to design an “encoder” a function that maps an image $x \in \mathcal{X}$ to a fixed-size vector that we can use as a compact representation of x . Next, we design one such encoder employing FFNNs and another employing CNNs.

```
[35]: class FlattenImage(nn.Module):
    def forward(self, input):
        return input.reshape(input.shape[:-3] + (-1,))

def build_ffnn_encoder(num_channels, width=64, height=64, output_size=1024, p_drop=0.):
    encoder = nn.Sequential(
        FlattenImage(),
        nn.Dropout(p_drop),
        nn.Linear(num_channels * width * height, output_size//2),
        nn.ReLU(),
        nn.Dropout(p_drop),
        nn.Linear(output_size//2, output_size//2),
        nn.ReLU(),
        nn.Dropout(p_drop),
        nn.Linear(output_size//2, output_size),
    )
    return encoder

def build_cnn_encoder(num_channels, width=64, height=64, output_size=1024, p_drop=0.):
    if width != 64:
        raise ValueError("The width is hardcoded")
    if height != 64:
        raise ValueError("The height is hardcoded")
    if output_size != 1024:
        raise ValueError("The output_size is hardcoded")
    # TODO: change the architecture so width, height and output_size are not hardcoded
    encoder = MySequential(
        nn.Conv2d(num_channels, 32, 4, 2),
        nn.LeakyReLU(0.2),
        nn.Conv2d(32, 64, 4, 2),
        nn.LeakyReLU(0.2),
        nn.Conv2d(64, 128, 4, 2),
        nn.LeakyReLU(0.2),
        nn.Conv2d(128, 256, 4, 2),
        nn.LeakyReLU(0.2),
        FlattenImage(),
        event_dims=3
```

(continues on next page)

(continued from previous page)

```
)  
    return encoder
```

[36]: # a batch of five [1, 64, 64]-dimensional images is encoded into
five 1024-dimensional vectors
build_ffnn_encoder(num_channels=1)(torch.zeros((5, 1, 64, 64))).shape
[36]: torch.Size([5, 1024])

[37]: # and, again, we can have structured batches
(here trying with (3,5))
build_ffnn_encoder(num_channels=1)(torch.zeros((3, 5, 1, 64, 64))).shape
[37]: torch.Size([3, 5, 1024])

[38]: # a batch of five [1, 64, 64]-dimensional images is encoded into
five 1024-dimensional vectors
build_cnn_encoder(num_channels=1)(torch.zeros((5, 1, 64, 64))).shape
[38]: torch.Size([5, 1024])

[39]: # and, again, since we use MySequential we can have structured batches
(here trying with (3,5))
build_cnn_encoder(num_channels=1)(torch.zeros((3, 5, 1, 64, 64))).shape
[39]: torch.Size([3, 5, 1024])

We can now design some CPD nets, assuming they map from an encoding of an image to a pmf over \mathcal{Z} .

Product of Bernoullis

This can be used to parameterise a cpd over binary vectors of fixed dimensionality.

OneHotCategorical

This can be used to parameterise a cpd over the one-hot encoding of categories from a finite set.

[40]: `class BernoulliCPDNet(CPDNet):
 """
 Output distribution is a product of Bernoulli distributions
 """

 def __init__(self, outcome_shape, num_inputs: int, hidden_size: int=None, p_drop:
 ↴float=0.):
 """
 outcome_shape: shape of the outcome (int or tuple)
 if int, we turn it into a singleton tuple
 num_inputs: rightmost dimensionality of the inputs to forward
 hidden_size: size of hidden layers for the CPDNet (use None to skip)
 p_drop: configure dropout before every Linear layer
 """
 super().__init__(outcome_shape)
 num_outputs = np.prod(self.outcome_shape)

 if hidden_size:`

(continues on next page)

(continued from previous page)

```

self.encoder = nn.Sequential(
    nn.Dropout(p_drop),
    nn.Linear(num_inputs, hidden_size),
    nn.ReLU()
)
else:
    self.encoder = nn.Identity()
hidden_size = num_inputs

self.logits = nn.Sequential(
    nn.Dropout(p_drop),
    nn.Linear(hidden_size, num_outputs),
    ReshapeLast(self.outcome_shape)
)

def forward(self, inputs):
    h = self.encoder(inputs)
    return td.Independent(td.Bernoulli(logits=self.logits(h)), len(self.outcome_
shape))

class CategoricalCPDNet(CPDNet):
    """
    Output distribution is a product of Bernoulli distributions
    """

    def __init__(self, outcome_shape, num_inputs: int, hidden_size: int=None, p_drop:_
float=0.):
        """
        outcome_shape: shape of the outcome (int or tuple)
            if int, we turn it into a singleton tuple
        num_inputs: rightmost dimensionality of the inputs to forward
        hidden_size: size of hidden layers for the CPDNet (use None to skip)
        p_drop: configure dropout before every Linear layer
        """
        super().__init__(outcome_shape)
        num_outputs = np.prod(self.outcome_shape)

        if hidden_size:
            self.encoder = nn.Sequential(
                nn.Dropout(p_drop),
                nn.Linear(num_inputs, hidden_size),
                nn.ReLU()
)
        else:
            self.encoder = nn.Identity()
            hidden_size = num_inputs

        self.logits = nn.Sequential(
            nn.Dropout(p_drop),
            nn.Linear(hidden_size, num_outputs),
            ReshapeLast(self.outcome_shape)
)

```

(continues on next page)

(continued from previous page)

```

    )

def forward(self, inputs):
    h = self.encoder(inputs)
    return td.OneHotCategorical(logits=self.logits(h))

```

```
[41]: def test_cpds(outcome_shape, batch_size=3, input_dim=5, hidden_size=2):

    cpd_net = BernoulliCPDNet(outcome_shape, num_inputs=input_dim, hidden_size=hidden_
    ↵size)
    print("\nBernoulli")
    print(cpd_net)
    print(f" outcome_shape={cpd_net.outcome_shape}")
    inputs = torch.from_numpy(np.random.uniform(size=(batch_size, input_dim))).float()
    print(f" shape of inputs: {inputs.shape}")
    p = cpd_net(inputs)
    print(f" distribution: {p}")
    z = p.sample()
    print(f" sample: {z}")
    print(f" shapes: sample={z.shape} log_prob={p.log_prob(z).shape}")

# Try a few
test_cpds(12)
test_cpds(12, hidden_size=None)
test_cpds((4, 5))
```

```

Bernoulli
BernoulliCPDNet(
    (encoder): Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=5, out_features=2, bias=True)
        (2): ReLU()
    )
    (logits): Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=2, out_features=12, bias=True)
        (2): ReshapeLast()
    )
)
outcome_shape=(12,)
shape of inputs: torch.Size([3, 5])
distribution: Independent(Bernoulli(logits: torch.Size([3, 12])), 1)
sample: tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
               [0., 1., 1., 0., 0., 1., 1., 0., 0., 1., 0., 1.],
               [1., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 1.]])
shapes: sample=torch.Size([3, 12]) log_prob=torch.Size([3])
```

```

Bernoulli
BernoulliCPDNet(
    (encoder): Identity()
```

(continues on next page)

(continued from previous page)

```
(logits): Sequential(
    (0): Dropout(p=0.0, inplace=False)
    (1): Linear(in_features=5, out_features=12, bias=True)
    (2): ReshapeLast()
)
)
outcome_shape=(12,)
shape of inputs: torch.Size([3, 5])
distribution: Independent(Bernoulli(logits: torch.Size([3, 12])), 1)
sample: tensor([[0., 0., 0., 0., 0., 0., 1., 1., 0., 0., 1.],
               [0., 0., 0., 1., 1., 0., 0., 1., 1., 0., 1.],
               [1., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0.]])
shapes: sample=torch.Size([3, 12]) log_prob=torch.Size([3])

Bernoulli
BernoulliCPDNet(
    (encoder): Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=5, out_features=2, bias=True)
        (2): ReLU()
    )
    (logits): Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=2, out_features=20, bias=True)
        (2): ReshapeLast()
    )
)
)
outcome_shape=(4, 5)
shape of inputs: torch.Size([3, 5])
distribution: Independent(Bernoulli(logits: torch.Size([3, 4, 5])), 2)
sample: tensor([[[1., 1., 0., 0., 1.],
                [0., 1., 1., 0., 0.],
                [0., 0., 0., 1., 1.],
                [1., 1., 1., 1., 1.]],

               [[1., 1., 0., 0., 0.],
                [0., 1., 1., 0., 1.],
                [0., 0., 1., 0., 1.],
                [1., 0., 0., 0., 1.]],

               [[1., 1., 1., 0., 1.],
                [0., 0., 0., 1., 0.],
                [1., 0., 1., 1., 1.],
                [1., 0., 0., 1., 0.]]])
shapes: sample=torch.Size([3, 4, 5]) log_prob=torch.Size([3])
```

Last, but certainly not least, we can combine our encoder and a choice of CPD net.

```
[42]: class InferenceModel(CPDNet):

    def __init__(
        self, cpd_net_type,
```

(continues on next page)

(continued from previous page)

```

latent_size, num_channels=1, width=64, height=64,
hidden_size=1024, p_drop=0.,
encoder_type=build_ffnn_encoder):

super().__init__(latent_size)

self.latent_size = latent_size
# encodes an image to a hidden_size-dimensional vector
self.encoder = encoder_type(
    num_channels=num_channels,
    width=width,
    height=height,
    output_size=hidden_size,
    p_drop=p_drop
)
# maps from a hidden_size-dimensional encoding
# to a cpd for Z|X=x
self.cpd_net = cpd_net_type(
    latent_size,
    num_inputs=hidden_size,
    hidden_size=2*latent_size,
    p_drop=p_drop
)

def forward(self, x):
    h = self.encoder(x)
    return self.cpd_net(h)

```

[43]: InferenceModel(BernoulliCPDNet, latent_size=10)(torch.zeros(5, 1, 64, 64))

[43]: Independent(Bernoulli(logits: torch.Size([5, 10])), 1)

We now have everything in place to use variational inference.

3.5 Neural Variational Inference

We will train our generative model via variational inference, for which we need to train an inference model along with it. We will see the ELBO objective, and gradient estimator based on score function estimation.

It's common to refer to any one such model as a variational auto-encoder (VAE), even though the original VAE was a very different generative model and employed a different gradient estimator.

Given a data point x , we estimate the gradient of the ELBO with respect to λ by MC estimating the following expression:

$$\nabla_{\lambda} \mathcal{E}(\lambda, \theta|x) = \mathbb{E} \left[r(z, x; \theta, \lambda) \nabla_{\lambda} \log \frac{p_{ZX}(z, x|\theta)}{q_{Z|X}(z|x, \lambda)} \right] \quad (4.73)$$

where the “reward” function is

$$r(z, x; \theta, \lambda) = \log p_{X|Z}(x|z, \theta) \quad (4.74)$$

And, because this gradient estimator is rather noisy, it's common to transform the reward function by further employing control variates. The simplest control variates are functions of x and possibly of θ and λ , but not a function of the action z with respect to which we evaluate the reward function. We will implement those as wrappers around the reward function. So, let's start by agreeing on the API of our control variates.

```
[44]: class VarianceReduction(nn.Module):
    """
    We will be using simple forms of control variates for variance reduction.
    These are transformations of the reward that are independent of the sampled
    latent variable, but they can, in principle, depend on x, and on the
    parameters of the generative and inference model.

    Some of these are trainable components, thus they also contribute to the loss.
    """

    def __init__(self):
        super().__init__()

    def forward(self, r, x, q, r_fn):
        """
        Return the transformed reward and a contribution to the loss.

        r: a batch of rewards
        x: a batch of observations
        q: policy
        r_fn: reward function
        """
        return r, torch.zeros_like(r)
```

Now we can work on our general NVIL model. The following class implements the NVIL objective as well as a lot of helper code to manipulate the model components in interesting ways (e.g., sampling, sampling conditionally, estimating marginal density, etc.)

```
[45]: class NVIL(nn.Module):
    """
    A generative model  $p(z)p(x|z)$  and an approximation  $q(z|x)$  to that
    model's true posterior.

    The approximation is estimated to maximise the ELBO, and so is the joint
    distribution.
    """

    def __init__(self, gen_model: JointDistribution, inf_model: InferenceModel, cv_model:
        ↪ VarianceReduction):
        """
        gen_model:  $p(z)p(x|z)$ 
        inf_model:  $q(z|x)$  which approximates  $p(z|x)$ 
        cv_model: optional transformations of the reward
        """
        super().__init__()
        self.gen_model = gen_model
        self.inf_model = inf_model
        self.cv_model = cv_model

    def gen_params(self):
        return self.gen_model.parameters()

    def inf_params(self):
```

(continues on next page)

(continued from previous page)

```

    return self.inf_model.parameters()

def cv_params(self):
    return self.cv_model.parameters()

def sample(self, batch_size, sample_size=None, oversample=False):
    """
    A sample from the joint distribution:
        z ~ prior
        x/z ~ obs model
    batch_size: number of samples in a batch
    sample_size: if None, the output tensor has shape [batch_size] + data_shape
                 if 1 or more, the output tensor has shape [sample_size, batch_size] + data_
    ↵shape
        while batch_size controls a parallel computation,
        sample_size controls a sequential computation (a for loop)
    oversample: if True, samples z (batch_size times), hold it fixed,
                and sample x (sample_size times)
    """
    pz = self.gen_model.prior((batch_size,))
    samples = [None] * (sample_size or 1)
    px_z = self.gen_model.obs_model(pz.sample()) if oversample else None
    for k in range(sample_size or 1):
        if not oversample:
            px_z = self.gen_model.obs_model(pz.sample())
            samples[k] = px_z.sample()
    x = torch.stack(samples)
    return x if sample_size else x.squeeze(0)

def cond_sample(self, x, sample_size=None, oversample=False):
    """
    Condition on x and draw a sample:
        z/x ~ inf model
        x'/z ~ obs model

    x: a batch of seed data samples
    sample_size: if None, the output tensor has shape [batch_size] + data_shape
                 if 1 or more, the output tensor has shape [sample_size, batch_size] + data_
    ↵shape
        sample_size controls a sequential computation (a for loop)
    oversample: if True, samples z (batch_size times), hold it fixed,
                and sample x' (sample_size times)
    """
    qz = self.inf_model(x)
    samples = [None] * (sample_size or 1)
    px_z = self.gen_model.obs_model(qz.sample()) if oversample else None
    for k in range(sample_size or 1):
        if not oversample:
            px_z = self.gen_model.obs_model(qz.sample())
            samples[k] = px_z.sample()
    x = torch.stack(samples)
    return x if sample_size else x.squeeze(0)

```

(continues on next page)

(continued from previous page)

```

def log_prob(self, z, x):
    """
    The log density of the joint outcome under the generative model
    z: [batch_size, latent_dim]
    x: [batch_size] + data_shape
    """
    return self.gen_model.log_prob(z=z, x=x)

def DRL(self, x, sample_size=None):
    """
    MC estimates of a model's
        * distortion D
        * rate R
        * and log-likelihood L
    The estimates are based on single data points
    but multiple latent samples.

    x: batch_shape + data_shape
    sample_size: if 1 or more, we use multiple samples
        sample_size controls a sequential computation (a for loop)
    """
    sample_size = sample_size or 1
    obs_dims = len(self.gen_model.cpd_net.outcome_shape)
    batch_shape = x.shape[:-obs_dims]
    with torch.no_grad():
        qz = self.inf_model(x)
        pz = self.gen_model.prior(batch_shape)
        R = td.kl_divergence(qz, pz)
        D = 0
        ratios = [None] * sample_size
        for k in range(sample_size):
            z = qz.sample()
            px_z = self.gen_model.obs_model(z)
            ratios[k] = pz.log_prob(z) + px_z.log_prob(x) - qz.log_prob(z)
            D = D - px_z.log_prob(x)
        ratios = torch.stack(ratios, dim=-1)
        L = torch.logsumexp(ratios, dim=-1) - np.log(sample_size)
        D = D / sample_size
    return D, R, L

def elbo(self, x, sample_size=None):
    """
    An MC estimate of ELBO = -D -R

    x: [batch_size] + data_shape
    sample_size: if 1 or more, we use multiple samples
        sample_size controls a sequential computation (a for loop)
    """
    D, R, _ = self.DRL(x, sample_size=sample_size)
    return -D -R

```

(continues on next page)

(continued from previous page)

```

def log_prob_estimate(self, x, sample_size=None):
    """
    An importance sampling estimate of  $\log p(x)$ 

    x: [batch_size] + data_shape
    sample_size: if 1 or more, we use multiple samples
        sample_size controls a sequential computation (a for loop)
    """
    _, _, L = self.DRL(x, sample_size=sample_size)
    return L

def forward(self, x, sample_size=None, rate_weight=1.):
    """
    A surrogate for an MC estimate of - grad ELBO

    x: [batch_size] + data_shape
    sample_size: if 1 or more, we use multiple samples
        sample_size controls a sequential computation (a for loop)
    cv: optional module for variance reduction
    """
    sample_size = sample_size or 1
    obs_dims = len(self.gen_model.cpd_net.outcome_shape)
    batch_shape = x.shape[:-obs_dims]

    qz = self.inf_model(x)
    pz = self.gen_model.prior(batch_shape)
    D = 0
    sfe = 0
    reward = 0
    cv_reward = 0
    raw_r = 0
    cv_loss = 0
    for _ in range(sample_size):
        z = qz.sample()
        px_z = self.gen_model.obs_model(z)
        raw_r = px_z.log_prob(x) + pz.log_prob(z) - qz.log_prob(z)
        r, l = self.cv_model(raw_r.detach(), x=x, q=qz, r_fn=lambda a: self.gen_
model(a).log_prob(x))
        cv_loss = cv_loss + l
        sfe = sfe + r.detach() * qz.log_prob(z)
        D = D - px_z.log_prob(x)
    D = (D / sample_size)
    sfe = (sfe / sample_size)
    R = td.kl_divergence(qz, pz)
    cv_loss = cv_loss / sample_size

    elbo_grad_surrogate = (-D + sfe)
    loss = -elbo_grad_surrogate + cv_loss

    return {'loss': loss.mean(0), 'ELBO': (-D -R).mean(0).item(), 'D': D.mean(0).
item(), 'R': R.mean(0).item(), 'cv_loss': cv_loss.mean(0).item()}

```

Here's an example

```
[46]: nvil = NVIL(
    JointDistribution(
        BernoulliPriorNet(10),
        BinarizedImageModel(
            num_channels=img_shape[0],
            width=img_shape[1],
            height=img_shape[2],
            latent_size=10,
            p_drop=0.1,
            decoder_type=build_ffnn_decoder
        )
    ),
    InferenceModel(
        cpd_net_type=BernoulliCPDNet,
        latent_size=10,
        num_channels=img_shape[0],
        width=img_shape[1],
        height=img_shape[2],
        encoder_type=build_ffnn_encoder
    ),
    VarianceReduction()
)
nvil
```

```
[46]: NVIL(
    (gen_model): JointDistribution(
        (prior_net): BernoulliPriorNet()
        (cpd_net): BinarizedImageModel(
            (decoder): Sequential(
                (0): Dropout(p=0.1, inplace=False)
                (1): Linear(in_features=10, out_features=512, bias=True)
                (2): ReLU()
                (3): Dropout(p=0.1, inplace=False)
                (4): Linear(in_features=512, out_features=512, bias=True)
                (5): ReLU()
                (6): Dropout(p=0.1, inplace=False)
                (7): Linear(in_features=512, out_features=4096, bias=True)
                (8): ReshapeLast()
            )
        )
    )
    (inf_model): InferenceModel(
        (encoder): Sequential(
            (0): FlattenImage()
            (1): Dropout(p=0.0, inplace=False)
            (2): Linear(in_features=4096, out_features=512, bias=True)
            (3): ReLU()
            (4): Dropout(p=0.0, inplace=False)
            (5): Linear(in_features=512, out_features=512, bias=True)
            (6): ReLU()
            (7): Dropout(p=0.0, inplace=False)
            (8): Linear(in_features=512, out_features=1024, bias=True)
        )
    )
)
```

(continues on next page)

(continued from previous page)

```
(cpd_net): BernoulliCPDNet(
    (encoder): Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=1024, out_features=20, bias=True)
        (2): ReLU()
    )
    (logits): Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=20, out_features=10, bias=True)
        (2): ReshapeLast()
    )
)
(cv_model): VarianceReduction()
)
```

[47]:

```
for x, y in train_loader:
    print('x.shape:', x.shape)
    print(nv1(x))
    break

x.shape: torch.Size([64, 1, 64, 64])
{'loss': tensor(-16754.9395, grad_fn=<MeanBackward1>), 'ELBO': -2844.470947265625, 'D': -2844.458740234375, 'R': 0.012489881366491318, 'cv_loss': 0.0}
```

3.5.1 Training algorithm

We have up to three components (recall that some control variates can have their own parameters), so we will be manipulating up to three optimisers:

[48]:

```
class OptCollection:

    def __init__(self, gen, inf, cv=None):
        self.gen = gen
        self.inf = inf
        self.cv = cv

    def zero_grad(self):
        self.gen.zero_grad()
        self.inf.zero_grad()
        if self.cv:
            self.cv.zero_grad()

    def step(self):
        self.gen.step()
        self.inf.step()
        if self.cv:
            self.cv.step()
```

Here's some helper code to assess and train the model

```
[49]: from collections import defaultdict, OrderedDict
from tqdm.auto import tqdm

def assess(model, sample_size, dl, device):
    """
    Wrapper for estimating a model's ELBO, distortion, rate, and log-likelihood
    using all data points in a data loader.
    """
    D = 0
    R = 0
    L = 0
    data_size = 0
    with torch.no_grad():
        for batch_x, batch_y in dl:
            Dx, Rx, Lx = model.DRL(batch_x.to(device), sample_size=sample_size)
            D = D + Dx.sum(0)
            R = R + Rx.sum(0)
            L = L + Lx.sum(0)
            data_size += batch_x.shape[0]
    D = D / data_size
    R = R / data_size
    L = L / data_size
    return {'ELBO': (-D -R).item(), 'D': D.item(), 'R': R.item(), 'L': L.item()}

def train_vae(model: NVIL, opts: OptCollection,
              training_data, dev_data,
              batch_size=64, num_epochs=10, check_every=10,
              sample_size_training=1,
              sample_size_eval=10,
              grad_clip=5.,
              num_workers=2,
              device=torch.device('cuda:0'))
):
    """
    model: pytorch model
    optimiser: pytorch optimiser
    training_corpus: a TaggedCorpus for trianing
    dev_corpus: a TaggedCorpus for dev
    batch_size: use more if you have more memory
    num_epochs: use more for improved convergence
    check_every: use less to check performance on dev set more often
    device: where we run the experiment

    Return a log of quantities computed during training (for plotting)
    """
    batcher = DataLoader(training_data, batch_size, shuffle=True, num_workers=num_workers,
                         pin_memory=True)
    dev_batcher = DataLoader(dev_data, batch_size, num_workers=num_workers, pin_memory=True)

    total_steps = num_epochs * len(batcher)
```

(continues on next page)

(continued from previous page)

```

log = defaultdict(list)

step = 0
model.eval()
for k, v in assess(model, sample_size_eval, dev_batcher, device=device).items():
    log[f"dev.{k}"].append((step, v))

with tqdm(range(total_steps)) as bar:
    for epoch in range(num_epochs):
        for batch_x, batch_y in batcher:
            model.train()
            opts.zero_grad()

            loss_dict = model(
                batch_x.to(device),
                sample_size=sample_size_training,
            )
            for metric, value in loss_dict.items():
                log[f'training.{metric}'].append((step, value))

            loss_dict['loss'].backward()

            nn.utils.clip_grad_norm_(
                model.parameters(),
                grad_clip
            )
            opts.step()

            bar_dict = OrderedDict()
            for metric, value in loss_dict.items():
                bar_dict[f'training.{metric}'] = f'{loss_dict[metric]:.2f}'
            for metric in ['ELBO', 'D', 'R', 'L']:
                bar_dict[f'dev.{metric}'] = f'{:.2f}'.format(log[f'dev.{metric}'][-1][1])
            bar.set_postfix(bar_dict)
            bar.update()

            if step % check_every == 0:
                model.eval()
                for k, v in assess(model, sample_size_eval, dev_batcher, device=device).items():
                    log[f"dev.{k}"].append((step, v))

            step += 1

model.eval()
for k, v in assess(model, sample_size_eval, dev_batcher, device=device).items():
    log[f"dev.{k}"].append((step, v))

return log

```

And, finally, some code to help inspect samples

```
[50]: def inspect_discrete_lvm(model, dl, device):
    for x, y in dl:

        x_ = model.sample(16, 4, oversample=True).cpu().reshape(-1, 1, 64, 64)
        plt.figure(figsize=(12,6))
        plt.axis('off')
        plt.imshow(make_grid(x_, nrow=16).permute((1, 2, 0)))
        plt.title("Prior samples")
        plt.show()

        plt.figure(figsize=(12,6))
        plt.axis('off')
        plt.imshow(make_grid(x, nrow=16).permute((1, 2, 0)))
        plt.title("Observations")
        plt.show()

        x_ = model.cond_sample(x.to(device)).cpu().reshape(-1, 1, 64, 64)
        plt.figure(figsize=(12,6))
        plt.axis('off')
        plt.imshow(make_grid(x_, nrow=16).permute((1, 2, 0)))
        plt.title("Conditional samples")
        plt.show()

    break
```

3.5.2 Variance reduction

Here are some concrete strategies for variance reduction. You can skip those in a first pass.

```
[51]: class CentredReward(VarianceReduction):
    """
    This control variate does not have trainable parameters,
    it maintains a running estimate of the average reward and updates
    a batch of rewards by computing reward - avg.
    """

    def __init__(self, alpha=0.9):
        super().__init__()
        self._alpha = alpha
        self._r_mean = 0.

    def forward(self, r, x=None, q=None, r_fn=None):
        """
        Centre the reward and update running estimates of mean.
        """
        with torch.no_grad():
            # sufficient statistics for next updates
            r_mean = torch.mean(r, dim=0)
            # centre the signal
            r = r - self._r_mean
            # update running estimate of mean
```

(continues on next page)

(continued from previous page)

```

self._r_mean = (1-self._alpha) * self._r_mean + self._alpha * r_mean.item()
return r, torch.zeros_like(r)

class ScaledReward(VarianceReduction):
    """
    This control variate does not have trainable parameters,
    it maintains a running estimate of the reward's standard deviation and
    updates a batch of rewards by computing reward / maximum(stddev, 1).
    """

    def __init__(self, alpha=0.9):
        super().__init__()
        self._alpha = alpha
        self._r_std = 1.0

    def forward(self, r, x=None, q=None, r_fn=None):
        """
        Scale the reward by a running estimate of std, and also update the estimate.
        """

        with torch.no_grad():
            # sufficient statistics for next updates
            r_std = torch.std(r, dim=0)
            # standardise the signal
            r = r / self._r_std
            # update running estimate of std
            self._r_std = (1-self._alpha) * self._r_std + self._alpha * r_std.item()
            # it's not safe to standardise with scales less than 1
            self._r_std = np.maximum(self._r_std, 1.)
        return r, torch.zeros_like(r)

class SelfCritic(VarianceReduction):
    """
    This control variate does not have trainable parameters,
    it updates a batch of rewards by computing reward - 'reward', where
    'reward' is (log p(X=x|Z=z')).detach() assessed for a novel sample
    z' ~ Z|X=x.
    """

    def __init__(self):
        super().__init__()

    def forward(self, r, x, q, r_fn):
        """
        Standardise the reward and update running estimates of mean/std.
        """

        with torch.no_grad():
            z = q.sample()
            r = r - r_fn(z, x)
        return r, torch.zeros_like(r)

```

(continues on next page)

(continued from previous page)

```

class Baseline(VarianceReduction):
    """
    An input-dependent baseline implemented as an MLP.
    The trainable parameters are adjusted via MSE.
    """

    def __init__(self, num_inputs, hidden_size, p_drop=0.):
        super().__init__()
        self.baseline = nn.Sequential(
            FlattenImage(),
            nn.Dropout(p_drop),
            nn.Linear(num_inputs, hidden_size),
            nn.ReLU(),
            nn.Dropout(p_drop),
            nn.Linear(hidden_size, 1)
        )

    def forward(self, r, x, q=None, r_fn=None):
        """
        Return r - baseline(x) and Baseline's loss.
        """

        # batch_shape + (1,)
        r_hat = self.baseline(x)
        # batch_shape
        r_hat = r_hat.squeeze(-1)
        loss = (r - r_hat)**2
        return r - r_hat.detach(), loss

class CVChain(VarianceReduction):

    def __init__(self, *args):
        super().__init__()
        if len(args) == 1 and isinstance(args[0], OrderedDict):
            for key, module in args[0].items():
                self.add_module(key, module)
        else:
            for idx, module in enumerate(args):
                self.add_module(str(idx), module)

    def forward(self, r, x, q, r_fn):
        loss = 0
        for cv in self._modules.values():
            r, l = cv(r, x=x, q=q, r_fn=r_fn)
            loss = loss + l
        return r, loss

```

3.5.3 Experiment

```
[52]: seed_all()

model = NVIL(
    JointDistribution(
        BernoulliPriorNet(10),
        BinarizedImageModel(
            num_channels=img_shape[0],
            width=img_shape[1],
            height=img_shape[2],
            latent_size=10,
            p_drop=0.1
        )
    ),
    InferenceModel(
        latent_size=10,
        num_channels=img_shape[0],
        width=img_shape[1],
        height=img_shape[2],
        cpd_net_type=BernoulliCPDNet
    ),
    VarianceReduction(), # no variance reduction
    # In the NVIL paper, this is what they use:
    # CVChain(
    #     CentredReward(),
    #     #Baseline(np.prod(img_shape), 512), # this is how you would use a trained_
    #     baselined
    #     # ScaledReward()
    # )
).to(my_device)

opts = OptCollection(
    opt.RMSprop(model.gen_params(), lr=5e-4, weight_decay=1e-6),
    opt.RMSprop(model.inf_params(), lr=1e-4),
    #opt.RMSprop(model.cv_params(), lr=1e-4, weight_decay=1e-6) # you need this if your_
    #baseline has trainable parameters
)

model

```

[52]: NVIL

```
(gen_model): JointDistribution(
(prior_net): BernoulliPriorNet()
(cpd_net): BinarizedImageModel(
(decoder): Sequential(
(0): Dropout(p=0.1, inplace=False)
(1): Linear(in_features=10, out_features=512, bias=True)
(2): ReLU()
(3): Dropout(p=0.1, inplace=False)
(4): Linear(in_features=512, out_features=512, bias=True)
(5): ReLU()
(6): Dropout(p=0.1, inplace=False)
```

(continues on next page)

(continued from previous page)

```

        (7): Linear(in_features=512, out_features=4096, bias=True)
        (8): ReshapeLast()
    )
)
)
(inf_model): InferenceModel(
    (encoder): Sequential(
        (0): FlattenImage()
        (1): Dropout(p=0.0, inplace=False)
        (2): Linear(in_features=4096, out_features=512, bias=True)
        (3): ReLU()
        (4): Dropout(p=0.0, inplace=False)
        (5): Linear(in_features=512, out_features=512, bias=True)
        (6): ReLU()
        (7): Dropout(p=0.0, inplace=False)
        (8): Linear(in_features=512, out_features=1024, bias=True)
    )
)
(cpd_net): BernoulliCPDNet(
    (encoder): Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=1024, out_features=20, bias=True)
        (2): ReLU()
    )
)
(logits): Sequential(
    (0): Dropout(p=0.0, inplace=False)
    (1): Linear(in_features=20, out_features=10, bias=True)
    (2): ReshapeLast()
)
)
)
(cv_model): VarianceReduction()
)

```

[53]:

```

log = train_vae(
    model=model,
    opts=opts,
    training_data=train_ds,
    dev_data=val_ds,
    batch_size=256,
    num_epochs=3, # use more for better models
    check_every=100,
    sample_size_training=1,
    sample_size_eval=1,
    grad_clip=5.,
    device=my_device
)
0%|          | 0/693 [00:00<?, ?it/s]

```

[54]:

```

log.keys()
dict_keys(['dev.ELBO', 'dev.D', 'dev.R', 'dev.L', 'training.loss', 'training.ELBO',
           'training.D', 'training.R', 'training.cv_loss'])

```

```
[55]: fig, axs = plt.subplots(1, 3 + int('training.cv_loss' in log), sharex=True, sharey=False,
                         figsize=(12, 3))

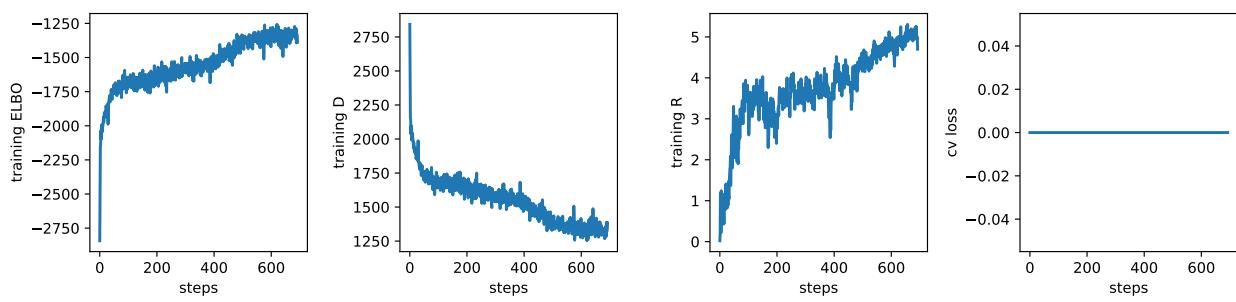
_ = axs[0].plot(np.array(log['training.ELBO'])[:,0], np.array(log['training.ELBO'])[:,1])
_= axs[0].set_ylabel("training ELBO")
_= axs[0].set_xlabel("steps")

_= axs[1].plot(np.array(log['training.D'])[:,0], np.array(log['training.D'])[:,1])
_= axs[1].set_ylabel("training D")
_= axs[1].set_xlabel("steps")

_= axs[2].plot(np.array(log['training.R'])[:,0], np.array(log['training.R'])[:,1])
_= axs[2].set_ylabel("training R")
_= axs[2].set_xlabel("steps")

if 'training.cv_loss' in log:
    _ = axs[3].plot(np.array(log['training.cv_loss'])[:,0], np.array(log['training.cv_loss'])[:,1])
    _ = axs[3].set_ylabel("cv loss")
    _ = axs[3].set_xlabel("steps")

fig.tight_layout(h_pad=1.2, w_pad=1.2)
```



```
[56]: fig, axs = plt.subplots(1, 4, sharex=True, sharey=False, figsize=(12, 3))

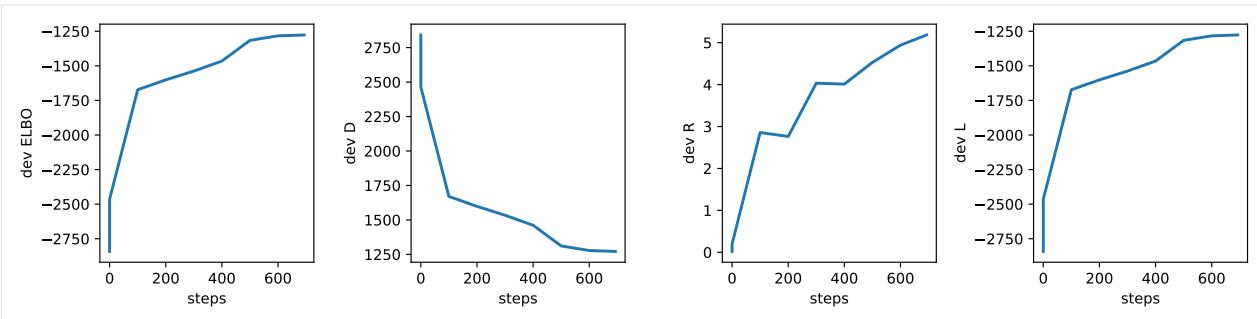
_= axs[0].plot(np.array(log['dev.ELBO'])[:,0], np.array(log['dev.ELBO'])[:,1])
_= axs[0].set_ylabel("dev ELBO")
_= axs[0].set_xlabel("steps")

_= axs[1].plot(np.array(log['dev.D'])[:,0], np.array(log['dev.D'])[:,1])
_= axs[1].set_ylabel("dev D")
_= axs[1].set_xlabel("steps")

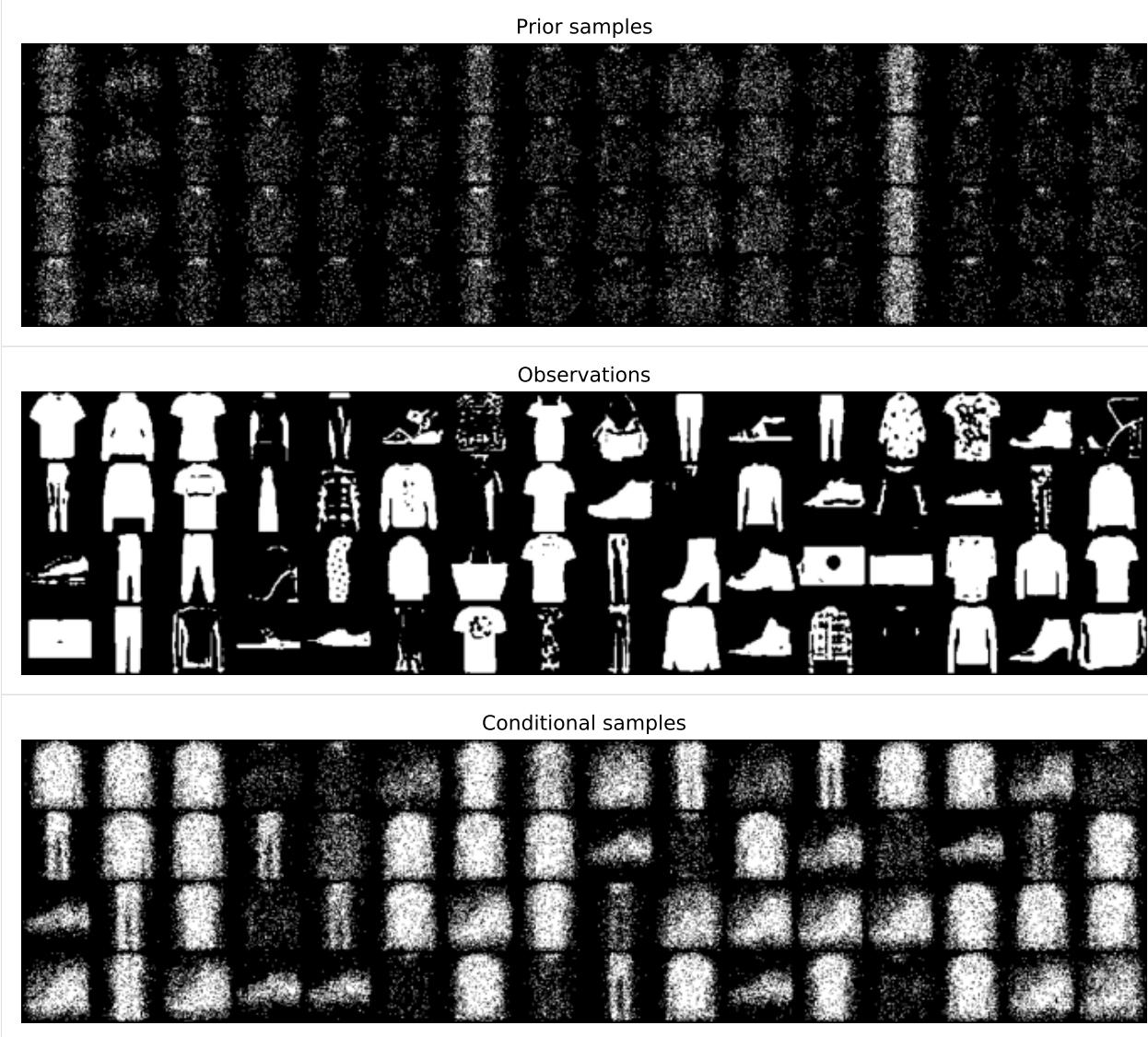
_= axs[2].plot(np.array(log['dev.R'])[:,0], np.array(log['dev.R'])[:,1])
_= axs[2].set_ylabel("dev R")
_= axs[2].set_xlabel("steps")

_= axs[3].plot(np.array(log['dev.L'])[:,0], np.array(log['dev.L'])[:,1])
_= axs[3].set_ylabel("dev L")
_= axs[3].set_xlabel("steps")

fig.tight_layout(h_pad=1.2, w_pad=1.2)
```



```
[57]: inspect_discrete_lvm(model, DataLoader(val_ds, 64, num_workers=2, pin_memory=True), my_
    ↴device)
```



4.33.5 4. Beyond

There are various things you can try.

- You can play with continuous output distributions and CNN encoder/decoders.
- Try switching baselines on and off.
- With VI, you should be able to use mixture models with larger K , even with the CNN decoder.
- You can change the dataset (e.g., use SVHN, whose images are coloured, thus you will be using 3 channels).
- You can attempt to (semi-)supervise the latent code (since these datasets offer some amount of supervision). For that you can use a log-likelihood side loss based on a model component that predicts a pmf $p_{Y|Z}(y|z, \phi)$ for the image's class given the latent code.

4.34 DPM 2 - Variational Inference for Deep Continuous LVMs

Filled notebook:

Authors: Wilker Aziz

4.34.1 0. Intended Learning Outcomes

After this tutorial the student should be able to

- parameterise a latent variable model with continuous latent variables
- estimate parameters using neural variational inference

Remark This tutorial builds upon the previous one and there is a lot of shared/unchanged code. The only changes are:

- additional prior nets (for continuous variables)
- additional CPD nets (for continuous variables)
- we changed DRL and forward in the NVIL class such that it can support LVMs trained via SFE or via reparameterisation

```
[1]: import torch
import numpy as np
import random
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
import torch.distributions as td

from functools import partial
from itertools import chain
from collections import defaultdict, OrderedDict
from tqdm.auto import tqdm
```

```
[2]: def seed_all(seed=42):
    np.random.seed(seed)
    random.seed(seed)
    torch.manual_seed(seed)
```

(continues on next page)

(continued from previous page)

```
seed_all()
```

4.34.2 1. Data

We are going to use toy image datasets for this notebook. These are fixed-dimensional observations for which encoder and decoders are relatively easy to design. This way we can focus on the aspects that are probabilistic in nature.

```
[3]: from torchvision.datasets import FashionMNIST
from torchvision import transforms
from torchvision.transforms import ToTensor
from torch.utils.data import random_split, Dataset
from torch.utils.data.dataloader import DataLoader
from torchvision.utils import make_grid
import torch.optim as opt

%matplotlib inline
```

A helper to binarize datasets:

```
[4]: class Binarizer(Dataset):

    def __init__(self, ds, threshold=0.5):
        self._ds = ds
        self._threshold = threshold

    def __len__(self):
        """Size of the corpus in number of sequence pairs"""
        return len(self._ds)

    def __getitem__(self, idx):
        """
        Return corpus_x[idx] and corpus_y[idx] converted to codes
        the latter has the EOS code in the end
        """
        x, y = self._ds[idx]
        return (x >= self._threshold).float(), y
```

FashionMNIST

```
[5]: dataset = FashionMNIST(root='data/', train=True, download=True,
                           transform=transforms.Compose([transforms.Resize(64), transforms.
                           ToTensor()]))
```

```
[6]: img_shape = dataset[0][0].shape
print("Shape of an image:", img_shape)
Shape of an image: torch.Size([1, 64, 64])
```

Let's make a dev set for ourselves:

```
[7]: val_size = 1000
train_size = len(dataset) - val_size
train_ds, val_ds = random_split(dataset, [train_size, val_size])
len(train_ds), len(val_ds)
```

```
[7]: (59000, 1000)
```

we suggest that you binarize the data in a first pass through this notebook, but as you will see, we can also model the continuous pixel intensities.

```
[8]: bin_data = True
```

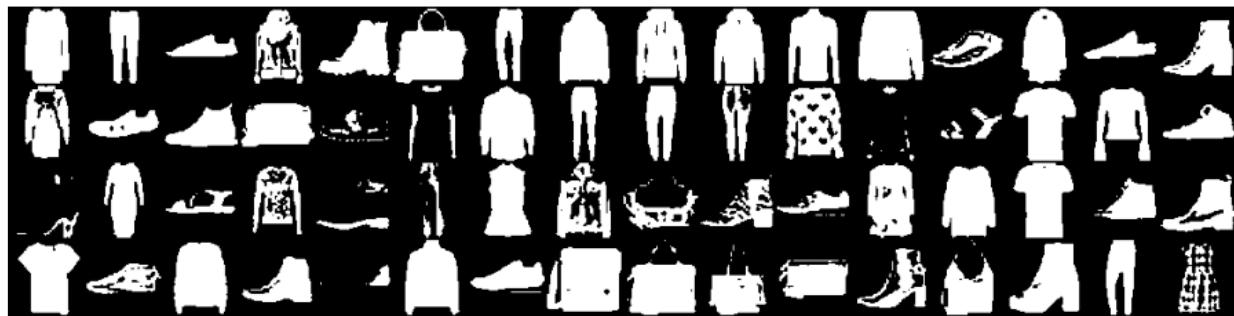
```
[9]: if bin_data:
    train_ds = Binarizer(train_ds)
    val_ds = Binarizer(val_ds)
```

```
[10]: batch_size = 64
train_loader = DataLoader(train_ds, batch_size, shuffle=True, num_workers=2, pin_
    _memory=True)
val_loader = DataLoader(val_ds, batch_size, num_workers=2, pin_memory=True)
```

Let's visualise a few samples

```
[11]: for images, y in train_loader:
    print('images.shape:', images.shape)
    plt.figure(figsize=(16,8))
    plt.axis('off')
    plt.imshow(make_grid(images, nrow=16).permute((1, 2, 0)))
    plt.show()
    break
```

```
images.shape: torch.Size([64, 1, 64, 64])
```



4.34.3 2. Latent variable models

We will be using NNs to parameterise a latent variable model, that is, a joint distribution over a collection of random variables (rvs), some of which are observed, some of which are not.

We are interested in two random variables (rvs):

- a discrete latent code $Z \in \mathcal{Z}$
- and an image $X \in \mathcal{X} \subseteq \mathbb{R}^D$

In this tutorial, x has a number C of channels, a certain width W and a certain height H , so $\mathcal{X} \subseteq \mathbb{R}^{C \times W \times H}$. Because we have fixed $D = C \times W \times H$, \mathcal{X} is finite-dimensional, but this need not be the case in general (for example, in a different domain, \mathcal{X} could be the unbounded space of all sentences of arbitrary length). We may treat the pixel intensities as discrete or continuous, as long as we choose an appropriate pmf/pdf for each case.

In this tutorial we will look into continuous latent codes. That is, $z \in \mathcal{Z} \subseteq \mathbb{R}^K$.

We specify a joint distribution over $\mathcal{Z} \times \mathcal{X}$ by specifying a joint probability density function (pdf):

$$p_{ZX}(z, x|\theta) = p_Z(z|\theta)p_{X|Z}(x|z, \theta) \quad (4.75)$$

Here θ denotes the parameters of the NNs that parameterise the pdf p_Z and the pdf $p_{X|Z=z}$ (for any given z).

In this tutorial, the prior is fixed, but in general it need not be. We do not have additional predictors to condition on, but in some application domains you may have (e.g., in imagine captioning, we may be interested in a joint for a caption y and a latent code z given an image x ; in image generation, we may be interested in a joint distribution for an image x and a latent code z given a caption y).

2.1 Prior networks

We begin by specifying the component that parameterises the prior p_Z .

A prior network is an NN that parameterises a fixed prior distribution for the instances in a batch.

```
[12]: class PriorNet(nn.Module):
    """
    An NN that parameterises a prior distribution.

    For this lab, our priors are fixed, so this NN's forward pass
    simply returns a fixed prior with a given batch_shape.
    """

    def __init__(self, outcome_shape: tuple):
        """
        outcome_shape: this is the shape of a single outcome
        if you use a single integer k, we will turn it into (k,)
        """
        super().__init__()
        if isinstance(outcome_shape, int):
            outcome_shape = (outcome_shape,)
        self.outcome_shape = outcome_shape

    def forward(self, batch_shape):
        """
        Returns a td object for the batch.
        """
        raise NotImplementedError("Implement me!"")
```

Let's implement two priors.

A standard Gaussian prior

Here the latent code is a point in the K -dimensional real coordinate space. We use a standard Gaussian per coordinate:

$$p_Z(z) = \prod_{k=1}^K \mathcal{N}(z_k | 0, 1) \quad (4.76)$$

A mixture of Gaussians prior

Here we learn a mixture of C Gaussians, each a product of K independent Gaussians:

$$p_Z(z|\theta) = \sum_{c=1}^C \omega_c \prod_{k=1}^K \mathcal{N}(z_k | \mu_c, \sigma_c^2) \quad (4.77)$$

where the prior parameters are the mixing coefficients $\omega_{1:C} \in \Delta_{C-1}$, the locations $\mu_{1:C} \in \mathbb{R}^C$ and the scales $\sigma_{1:C} \in \mathbb{R}_{>0}^C$.

```
[13]: class GaussianPriorNet(PriorNet):
    """
    For z a K-dimensional code:

    p(z) = prod_k Normal(z[k]/0, 1)
    """

    def __init__(self, outcome_shape):
        super().__init__(outcome_shape)
        # the product of Bernoulli priors will have Bernoulli(0.5) factors
        self.register_buffer("locs", torch.zeros(self.outcome_shape, requires_grad=False).detach())
        self.register_buffer("scales", torch.ones(self.outcome_shape, requires_grad=False).detach())

    def forward(self, batch_shape):
        shape = batch_shape + self.outcome_shape
        # we wrap around td.Independent to obtain a pdf over multivariate draws
        return td.Independent(td.Normal(loc=self.locs.expand(shape), scale=self.scales.expand(shape)), len(self.outcome_shape))

class MoGPriorNet(PriorNet):
    """
    For z a K-dimensional code:

    p(z|w_1...w_C, u_1...u_C, s_1...s_C)
    = \sum_c w_c prod_k Normal(z[k]/u[c], s[c]^2)
    """

    def __init__(self, outcome_shape, num_components, lbound=-10, rbound=10):
        super().__init__(outcome_shape)
        # [C]
        self.logits = nn.Parameter(torch.rand(num_components, requires_grad=True), requires_grad=True)
        # (C,) + outcome_shape
```

(continues on next page)

(continued from previous page)

```

shape = (num_components,) + self.outcome_shape
self.locs = nn.Parameter(torch.rand(shape, requires_grad=True), requires_
grad=True)
self.scales = nn.Parameter(1 + torch.rand(shape, requires_grad=True), requires_
grad=True)
self.num_components = num_components

def forward(self, batch_shape):
    # e.g., with batch_shape (B,) and outcome_shape (K,) this is
    # [B, C, K]
    shape = batch_shape + (self.num_components,) + self.outcome_shape
    # we wrap around td.Independent to obtain a pdf over multivariate draws
    # (note that C is not part of the event_shape, thus td.Independent will
    # should not treat that dimension as part of the outcome)
    # in our example, a draw from independent would return [B, C] draws of K-
    # dimensional outcomes
    comps = td.Independent(td.Normal(loc=self.locs.expand(shape), scale=self.scales.
expand(shape), len(self.outcome_shape))
        # a batch of component selectors
        pc = td.Categorical(logits=self.logits.expand(batch_shape + (self.num_components,
)))
    # and finally, a mixture
    return td.MixtureSameFamily(pc, comps)

```

[14]: def test_priors(batch_size=2, latent_dim=3, num_comps=5):

```

prior_net = GaussianPriorNet(latent_dim)
print("\nGaussian")
print(" trainable parameters")
print(list(prior_net.parameters()))
print(f" outcome_shape={prior_net.outcome_shape}")
p = prior_net(batch_shape=(batch_size,))
print(f" distribution: {p}")
z = p.sample()
print(f" sample: {z}")
print(f" shapes: sample={z.shape} log_prob={p.log_prob(z).shape}")

prior_net = MoGPriorNet(latent_dim, num_comps)
print("\nMixture of Gaussian")
print(" trainable parameters")
print(list(prior_net.parameters()))
print(f" outcome_shape={prior_net.outcome_shape}")
p = prior_net(batch_shape=(batch_size,))
print(f" distribution: {p}")
z = p.sample()
print(f" sample: {z}")
print(f" shapes: sample={z.shape} log_prob={p.log_prob(z).shape}")

```

test_priors()

(continues on next page)

(continued from previous page)

```
Gaussian
  trainable parameters
[]
outcome_shape=(3,)
distribution: Independent(Normal(loc: torch.Size([2, 3]), scale: torch.Size([2, 3])), 1)
sample: tensor([-0.3545, -0.5105, -0.7530],
              [ 0.6498, -0.0851, -1.2621])
shapes: sample=torch.Size([2, 3]) log_prob=torch.Size([2])

Mixture of Gaussian
  trainable parameters
[Parameter containing:
tensor([0.8617, 0.8520, 0.3585, 0.6196, 0.5566], requires_grad=True), Parameterred
↳ containing:
tensor([[0.4819, 0.0711, 0.2805],
       [0.4312, 0.1763, 0.3839],
       [0.0172, 0.8007, 0.8341],
       [0.6358, 0.9348, 0.1698],
       [0.6220, 0.4291, 0.3030]], requires_grad=True), Parameter containing:
tensor([[1.5164, 1.3117, 1.3240],
       [1.5596, 1.3319, 1.5549],
       [1.1613, 1.6315, 1.0815],
       [1.9174, 1.9954, 1.5884],
       [1.8620, 1.4976, 1.3426]], requires_grad=True)]
outcome_shape=(3,)
distribution: MixtureSameFamily(
  Categorical(logits: torch.Size([2, 5])),
  Independent(Normal(loc: torch.Size([2, 5, 3]), scale: torch.Size([2, 5, 3])), 1))
sample: tensor([-1.5431, -0.7577,  0.3158],
              [ 3.9455, -0.1169,  0.7572])
shapes: sample=torch.Size([2, 3]) log_prob=torch.Size([2])
```

2.2 Conditional probability distributions

Next, we create code to parameterise conditional probability distributions (cpds), which we do by having an NN parameterise a choice of pmf/pdf. This will be useful in parameterising the $p_{X|Z=z}$ component of our latent variable models (and, later on, it will also be useful for variational inference, when we develop $q_{Z|X=x}$).

Our general strategy is to map from a number of inputs (which the user will choose) to the parameters of a pmf/pdf supported by `torch.distributions`.

```
[15]: class CPDNet(nn.Module):
    """
    Let L be a choice of distribution
    and x ~ L is an outcome with shape outcome_shape

    This is an NN whose forward method maps from a number of inputs to the
    parameters of L's pmf/pdf and returns a torch.distributions
    object representing L's pmf/pdf.
    """

```

(continues on next page)

(continued from previous page)

```

def __init__(self, outcome_shape):
    """
        outcome_shape: this is the shape of a single outcome
        if you use a single integer k, we will turn it into (k,)
    """
    super().__init__()
    if isinstance(outcome_shape, int):
        outcome_shape = (outcome_shape,)
    self.outcome_shape = outcome_shape

def forward(self, inputs):
    """
        Return a torch.distribution object predicted from `inputs`.
    """
    inputs: a tensor with shape batch_shape + (num_inputs,)
    """
    raise NotImplementedError("Implemented me")

```

2.2.1 Observational model

The observational model prescribes the distribution of $X|Z = z$.

If we assume our pixel intensities are binary, we can use a product of $C \times W \times H$ Bernoulli distributions, which we parameterise jointly using an NN:

$$p_{X|Z}(x|z, \theta) = \prod_{c=1}^C \prod_{w=1}^W \prod_{h=1}^H \text{Bernoulli}(x_{c,w,h}|f_{c,w,h}(z; \theta)) \quad (4.78)$$

Here $\mathbf{f}(z; \theta) \in (0, 1)^C \times (0, 1)^W \times (0, 1)^H$ is an NN architecture such as a feed-forward net or a stack of transposed convolution layers. In NN literature, such architectures are often called *decoders*.

If we assume our pixel intensities are real values in $[0, 1]$ (0 and 1 included), we need to parameterise a pdf. A good choice of pdf is the [ContinuousBernoulli distributions](#), which is a single-parameter distribution (much like the Bernoulli) whose support is the set $[0, 1]$.

Let's start by designing \mathbf{f} .

A very basic design uses a FFNN:

```

[16]: class ReshapeLast(nn.Module):
    """
        Helper layer to reshape the rightmost dimension of a tensor.

        This can be used as a component of nn.Sequential.
    """

    def __init__(self, shape: tuple):
        """
            shape: desired rightmost shape
        """
        super().__init__()
        self.shape = shape

```

(continues on next page)

(continued from previous page)

```

def forward(self, input):
    # reshapes the last dimension into self.shape
    return input.reshape(input.shape[:-1] + self._shape)

def build_ffnn_decoder(latent_size, num_channels, width=64, height=64, hidden_size=512, ↴
→ p_drop=0.):
    """
    Map the latent code to a tensor with shape [num_channels, width, height]
    using a FFNN with 2 hidden layers.

    latent_size: size of latent code
    num_channels: number of channels in the output
    width: image shape
    height: image shape
    hidden_size: we first map from latent_size to hidden_size and
        then use feed forward NNs to map it to [num_channels, width, height]
    p_drop: dropout rate before linear layers
    """

    decoder = nn.Sequential(
        nn.Dropout(p_drop),
        nn.Linear(latent_size, hidden_size),
        nn.ReLU(),
        nn.Dropout(p_drop),
        nn.Linear(hidden_size, hidden_size),
        nn.ReLU(),
        nn.Dropout(p_drop),
        nn.Linear(hidden_size, num_channels * width * height),
        ReshapeLast((num_channels, width, height)),
    )
    return decoder

```

[17]: # mapping from 10-dimensional latent code
`build_ffnn_decoder(latent_size=10, num_channels=1)(torch.zeros((5, 10))).shape`
[17]: `torch.Size([5, 1, 64, 64])`

[18]: # we can also have a structured batch shape (e.g., [3, 5])
`build_ffnn_decoder(latent_size=10, num_channels=1)(torch.zeros((3, 5, 10))).shape`
[18]: `torch.Size([3, 5, 1, 64, 64])`

The downside is that the output layer is rather large.

An architecture with inductive biases that are more appropriate for our data type is a CNN, in particular, a transposed CNN. Here we design one such decoder:

[19]: `class MySequential(nn.Sequential):`
 """
This is a version of nn.Sequential that works with structured batches
(i.e., batches that have multiple dimensions)
even when some of the nn layers in it does not.

(continues on next page)

(continued from previous page)

The idea is to just wrap `nn.Sequential` around two calls to `reshape` which remove and restore the batch dimensions.

```

"""
def __init__(self, *args, event_dims=1):
    super().__init__(*args)
    self._event_dims = event_dims

def forward(self, input):
    # memorise batch shape
    batch_shape = input.shape[:-self._event_dims]
    # memorise latent shape
    event_shape = input.shape[-self._event_dims:]
    # flatten batch shape and obtain outputs
    output = super().forward(input.reshape((-1,) + event_shape))
    # restore batch shape
    return output.reshape(batch_shape + output.shape[1:])

def build_cnn_decoder(latent_size, num_channels, width=64, height=64, hidden_size=1024, ↴
    p_drop=0.):
"""
Map the latent code to a tensor with shape [num_channels, width, height].

latent_size: size of latent code
num_channels: number of channels in the output
width: must be 64 (for now)
height: must be 64 (for now)
hidden_size: we first map from latent_size to hidden_size and
    then use transposed 2d convolutions to [num_channels, width, height]
p_drop: dropout rate before linear layers
"""
if width != 64:
    raise ValueError("The width is hardcoded")
if height != 64:
    raise ValueError("The height is hardcoded")

# TODO: change the architecture so width and height are not hardcoded
decoder = MySequential(
    nn.Dropout(p_drop),
    nn.Linear(latent_size, hidden_size),
    ReshapeLast((hidden_size, 1, 1)),
    nn.ConvTranspose2d(hidden_size, 128, 5, 2),
    nn.ReLU(),
    nn.ConvTranspose2d(128, 64, 5, 2),
    nn.ReLU(),
    nn.ConvTranspose2d(64, 32, 6, 2),
    nn.ReLU(),
    nn.ConvTranspose2d(32, num_channels, 6, 2),
    event_dims=1
)
return decoder

```

```
[20]: # a batch of five 10-dimensional latent codes is transformed
# into a batch of 5 images, each with shape [1, 64, 64]
build_cnn_decoder(latent_size=10, num_channels=1)(torch.zeros((5, 10))).shape
[20]: torch.Size([5, 1, 64, 64])

[21]: # note that because we use MySequential,
# we can have a batch of [3, 5] assignments
# (this is useful, for example, when we have multiple draws of the latent
# variable for each of the data points in the batch)
build_cnn_decoder(latent_size=10, num_channels=1)(torch.zeros((3, 5, 10))).shape
[21]: torch.Size([3, 5, 1, 64, 64])
```

Now we are in position to design a CPDNet for our image model, it simply combines a choice of decoder with a choice of distribution:

```
[22]: class BinarizedImageModel(CPDNet):

    def __init__(self, num_channels, width, height, latent_size, decoder_type=build_ffnn_
decoder, p_drop=0.):
        super().__init__((num_channels, width, height))
        self.decoder = decoder_type(
            latent_size=latent_size,
            num_channels=num_channels,
            width=width,
            height=height,
            p_drop=p_drop
        )

    def forward(self, z):
        """
        Return the cpd X/Z=z

        z: batch_shape + (latent_dim,)
        """
        # batch_shape + (num_channels, width, height)
        h = self.decoder(z)
        return td.Independent(td.Bernoulli(logits=h), len(self.outcome_shape))
```

```
class ContinuousImageModel(CPDNet):
    # TODO: this could be an exercise

    def __init__(self, num_channels, width, height, latent_size, decoder_type=build_ffnn_
decoder, p_drop=0.):
        super().__init__((num_channels, width, height))
        self.decoder = decoder_type(
            latent_size=latent_size,
            num_channels=num_channels,
            width=width,
            height=height,
            p_drop=p_drop
        )
```

(continues on next page)

(continued from previous page)

```
def forward(self, z):
    """
    Return the cpd X/Z=z

    z: batch_shape + (latent_dim,)
    """
    # batch_shape + (num_channels, width, height)
    h = self.decoder(z)
    return td.Independent(td.ContinuousBernoulli(logits=h), len(self.outcome_shape))
```

[23]:

```
obs_model = BinarizedImageModel(
    num_channels=img_shape[0],
    width=img_shape[1],
    height=img_shape[2],
    latent_size=10,
    p_drop=0.1,
)
print(obs_model)
# a batch of five zs is mapped to 5 distributions over [1,64,64]-dimensional
# binary tensors
print(obs_model(torch.zeros([5, 10])))
```

```
BinarizedImageModel(
    decoder: Sequential(
        (0): Dropout(p=0.1, inplace=False)
        (1): Linear(in_features=10, out_features=512, bias=True)
        (2): ReLU()
        (3): Dropout(p=0.1, inplace=False)
        (4): Linear(in_features=512, out_features=512, bias=True)
        (5): ReLU()
        (6): Dropout(p=0.1, inplace=False)
        (7): Linear(in_features=512, out_features=4096, bias=True)
        (8): ReshapeLast()
    )
)
Independent(Bernoulli(logits: torch.Size([5, 1, 64, 64])), 3)
```

We can also use a different decoder

[24]:

```
obs_model = BinarizedImageModel(
    num_channels=img_shape[0],
    width=img_shape[1],
    height=img_shape[2],
    latent_size=10,
    p_drop=0.1,
    decoder_type=build_cnn_decoder
)
print(obs_model)
# a batch of five zs is mapped to 5 distributions over [1,64,64]-dimensional
# binary tensors
print(obs_model(torch.zeros([5, 10])))
```

```

BinarizedImageModel(
    decoder: MySequential(
        (0): Dropout(p=0.1, inplace=False)
        (1): Linear(in_features=10, out_features=1024, bias=True)
        (2): ReshapeLast()
        (3): ConvTranspose2d(1024, 128, kernel_size=(5, 5), stride=(2, 2))
        (4): ReLU()
        (5): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2))
        (6): ReLU()
        (7): ConvTranspose2d(64, 32, kernel_size=(6, 6), stride=(2, 2))
        (8): ReLU()
        (9): ConvTranspose2d(32, 1, kernel_size=(6, 6), stride=(2, 2))
    )
)
Independent(Bernoulli(logits: torch.Size([5, 1, 64, 64])), 3)

```

2.3 Joint distribution

We can now combine a prior and an observational model into a joint distribution. A joint distribution supports a few important operations such as marginal and posterior pdf assessments, as well as sampling from the joint distribution. Marginal and posterior assessments require computations that may or may not be tractable, see below.

From a joint pdf, we can compute the marginal density of x via

$$p_X(x|\theta) = \int_{\mathcal{Z}} p_{ZX}(z, x|\theta) dz \quad (4.79)$$

$$= \int_{\mathcal{Z}} p_Z(z|\theta) p_{X|Z}(x|z, \theta) dz \quad (4.80)$$

For uncountable \mathcal{Z} and a general enough parameterisation of $p_{X|Z=z}$, this is intractable (a special case where this is tractable is that where $p_{X|Z=z}$ is itself a Gaussian whose mean depend linearly on z , such a conditional is unreasonably simple for interesting datasets).

The posterior density of z given x depends on the intractable marginal:

$$p_{Z|X}(z|x, \theta) = \frac{p_Z(z|\theta) p_{X|Z}(x|z, \theta)}{p_X(x|\theta)} \quad (4.81)$$

As marginalisation is intractable, we can obtain a naive lowerbound by direct application of Jensen's inequality:

$$\log p_X(x|\theta) = \log \int_{\mathcal{Z}} p_Z(z|\theta) p_{X|Z}(x|z, \theta) dz \quad (4.82)$$

$$\stackrel{\text{JI}}{\geq} \int_{\mathcal{Z}} p_Z(z|\theta) \log p_{X|Z}(x|z, \theta) dz \quad (4.83)$$

$$\stackrel{\text{MC}}{\approx} \frac{1}{S} \sum_{s=1}^S \log p_{X|Z}(x|z_s, \theta) \quad \text{where } z_s \sim p_Z \quad (4.84)$$

A better lowerbound could be obtained via importance sampling, but it would require training an approximating distribution (as we will do in variational inference).

Recall that, given a dataset \mathcal{D} , the log-likelihood function $\mathcal{L}(\theta|\mathcal{D}) = \sum_{x \in \mathcal{D}} \log p_X(x|\theta)$ requires performing marginal density assessments. Whenever exact marginalisation is intractable, we are unable to assess $\mathcal{L}(\theta|\mathcal{D})$ and its gradient with respect to θ . If the prior is fixed, we can use the naive lowerbound to obtain a gradient estimate, but, again, our naive application of JI leads to a generally rather loose bound.

```
[25]: class JointDistribution(nn.Module):
    """
    A wrapper to combine a prior net and a cpd net into a joint distribution.
    """

    def __init__(self, prior_net: PriorNet, cpd_net: CPDNet):
        """
        prior_net: object to parameterise p_Z
        cpd_net: object to parameterise p_{X|Z=z}
        """
        super().__init__()
        self.prior_net = prior_net
        self.cpd_net = cpd_net

    def prior(self, shape):
        return self.prior_net(shape)

    def obs_model(self, z):
        return self.cpd_net(z)

    def sample(self, shape):
        """
        Return z via prior_net(shape).sample()
        and x via cpd_net(z).sample()
        """
        pz = self.prior_net(shape)
        z = pz.sample()
        px_z = self.cpd_net(z)
        x = px_z.sample()
        return z, x

    def log_prob(self, z, x):
        """
        Assess the log density of the joint outcome.
        """
        batch_shape = z.shape[:-len(self.prior_net.outcome_shape)]
        pz = self.prior_net(batch_shape)
        px_z = self.cpd_net(z)
        return pz.log_prob(z) + px_z.log_prob(x)

    def log_marginal(self, x, enumerate_fn):
        """
        Return log marginal density of x.

        enumerate_fn: function that enumerates the support of the prior
        (this is needed for marginalisation  $p(x) = \int p(z, x) dz$ )
        """
        This only really makes sense if the support is a
        (small) countably finite set. In such cases, you can use
        enumerate=lambda p: p.enumerate_support()
        which is supported, for example, by Categorical and OneHotCategorical.

        If the support is discrete (eg, bit vectors) you can still dare to
        """

```

(continues on next page)

(continued from previous page)

enumerate it explicitly, but you will need to write customised code, as torch.distributions will not offer that functionality for you.

If the support is uncountable, countably infinite, or just large anyway, you need approximate tools (such as VI, importance sampling, etc)

```

batch_shape = x.shape[:-len(self.cpd_net.outcome_shape)]
pz = self.prior_net(batch_shape)
log_joint = []
# (support_size,) + batch_shape
z = enumerate_fn(pz)
px_z = self.cpd_net(z)
# (support_size,) + batch_shape
log_joint = pz.log_prob(z) + px_z.log_prob(x.unsqueeze(0))
# batch_shape
return torch.logsumexp(log_joint, 0)

def posterior(self, x, enumerate_fn):
    """
    Return the posterior distribution Z|X=x.

    As the code is discrete, we return a discrete distribution over
    the complete space of all possible latent codes. This is done via
    exhaustive enumeration provided by `enumerate_fn`.
    """
    batch_shape = x.shape[:-len(self.cpd_net.outcome_shape)]
    pz = self.prior_net(batch_shape)
    # (support_size,) + batch_shape
    z = enumerate_fn(pz)
    px_z = self.cpd_net(z)
    # (support_size,) + batch_shape
    log_joint = pz.log_prob(z) + px_z.log_prob(x.unsqueeze(0))
    # batch_shape + (support_size,)
    log_joint = torch.stack(log_joint, 0, -1)
    return td.Categorical(logits=log_joint)

def naive_lowerbound(self, x, num_samples: int):
    """
    Return an MC lowerbound on log marginal density of x:
        log p(x) >= 1/S \sum_s log p(x|z[s])
        with z[s] ~ p_Z
    """
    batch_shape = x.shape[:-len(self.cpd_net.outcome_shape)]
    pz = self.prior_net(batch_shape)
    # (num_samples,) + batch_shape + prior_outcome_shape
    log_probs = []
    # I'm using a for loop, but note that with enough GPU memory
    # one could parallelise this step
    for z in pz.sample((num_samples,)):
        px_z = self.cpd_net(z)
        log_probs.append(px_z.log_prob(x))
    # (num_samples,) + batch_shape

```

(continues on next page)

(continued from previous page)

```
log_probs = torch.stack(log_probs)
# batch_shape
return torch.mean(log_probs, 0)
```

```
[26]: def test_joint_dist(latent_size=10, num_comps=3, data_shape=(1, 64, 64), batch_size=2, ↴
    ↴hidden_size=32):
    p = JointDistribution(
        prior_net=GaussianPriorNet(latent_size),
        cpd_net=BinarizedImageModel(
            num_channels=data_shape[0],
            width=data_shape[1],
            height=data_shape[2],
            latent_size=latent_size,
            decoder_type=build_cnn_decoder
        )
    )
    print("Model for binarized data")
    print(p)
    z, x = p.sample((batch_size,))
    print("sampled z")
    print(z)
    print("sampled x")
    print(x)
    print("MC lowerbound")
    print(" 1:", p.naive_lowerbound(x, 10))
    print(" 2:", p.naive_lowerbound(x, 10))

    print("\n\n")

    p = JointDistribution(
        prior_net=MoGPriorNet(latent_size, num_comps),
        cpd_net=ContinuousImageModel(
            num_channels=data_shape[0],
            width=data_shape[1],
            height=data_shape[2],
            latent_size=latent_size,
            decoder_type=build_cnn_decoder
        )
    )
    print("Model for continuous data")
    print(p)
    z, x = p.sample((batch_size,))
    print("sampled z")
    print(z)
    print("sampled x")
    print(x)
    print("MC lowerbound")
    print(" 1:", p.naive_lowerbound(x, 10))
    print(" 2:", p.naive_lowerbound(x, 10))

test_joint_dist(10)
```

```

Model for binarized data
JointDistribution(
    (prior_net): GaussianPriorNet()
    (cpd_net): BinarizedImageModel(
        (decoder): MySequential(
            (0): Dropout(p=0.0, inplace=False)
            (1): Linear(in_features=10, out_features=1024, bias=True)
            (2): ReshapeLast()
            (3): ConvTranspose2d(1024, 128, kernel_size=(5, 5), stride=(2, 2))
            (4): ReLU()
            (5): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2))
            (6): ReLU()
            (7): ConvTranspose2d(64, 32, kernel_size=(6, 6), stride=(2, 2))
            (8): ReLU()
            (9): ConvTranspose2d(32, 1, kernel_size=(6, 6), stride=(2, 2))
        )
    )
)
sampled z
tensor([-0.8407,  0.2065, -1.2518, -0.9531,  0.1592, -0.1754, -1.6121,  0.1075,
        -0.7088, -0.3303],
       [-0.1712,  0.1613, -0.3089,  0.2564,  0.0636,  0.7447,  0.7566, -0.4789,
        1.1534, -0.8448]])
sampled x
tensor([[[[0., 1., 1., ..., 1., 1., 1.],
          [1., 1., 0., ..., 1., 0., 1.],
          [1., 0., 1., ..., 0., 0., 1.],
          ...,
          [0., 0., 1., ..., 0., 1., 0.],
          [1., 1., 0., ..., 1., 1., 1.],
          [1., 0., 0., ..., 0., 0., 1.]]],
       [[[0., 1., 0., ..., 1., 1., 1.],
          [1., 1., 0., ..., 1., 0., 0.],
          [0., 0., 0., ..., 1., 0., 1.],
          ...,
          [0., 1., 0., ..., 1., 1., 1.],
          [1., 1., 0., ..., 1., 0., 0.],
          [0., 0., 1., ..., 0., 1., 1.]]]])
MC lowerbound
1: tensor([-2837.7363, -2837.9871], grad_fn=<MeanBackward1>)
2: tensor([-2837.9243, -2838.0288], grad_fn=<MeanBackward1>)

```

```

Model for continuous data
JointDistribution(
    (prior_net): MoGPriorNet()
    (cpd_net): ContinuousImageModel(
        (decoder): MySequential(
            (0): Dropout(p=0.0, inplace=False)
            (1): Linear(in_features=10, out_features=1024, bias=True)

```

(continues on next page)

(continued from previous page)

```

(2): ReshapeLast()
(3): ConvTranspose2d(1024, 128, kernel_size=(5, 5), stride=(2, 2))
(4): ReLU()
(5): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2))
(6): ReLU()
(7): ConvTranspose2d(64, 32, kernel_size=(6, 6), stride=(2, 2))
(8): ReLU()
(9): ConvTranspose2d(32, 1, kernel_size=(6, 6), stride=(2, 2))
)
)
)
)
sampled z
tensor([[ 0.5024,  0.1232,  1.0614,  1.5447,  1.1263,  1.0789,  2.7709,  1.9042,
         -1.0329,  3.2964],
        [ 1.0950, -1.1369,  1.0845,  1.6282, -0.8367,  1.1774, -0.4392, -0.3845,
         1.5312, -1.0509]])
sampled x
tensor([[[[0.2456, 0.6908, 0.2022, ..., 0.9125, 0.4140, 0.0154],
          [0.9211, 0.8082, 0.1273, ..., 0.0145, 0.6978, 0.2584],
          [0.9423, 0.6667, 0.8885, ..., 0.8678, 0.2550, 0.1817],
          ...,
          [0.9706, 0.4656, 0.9801, ..., 0.3806, 0.5779, 0.3176],
          [0.8841, 0.9396, 0.1979, ..., 0.4812, 0.1924, 0.1274],
          [0.9042, 0.7917, 0.8952, ..., 0.4248, 0.3569, 0.6764]]],
        [[[0.2313, 0.6762, 0.1603, ..., 0.8595, 0.0705, 0.6296],
          [0.6333, 0.8830, 0.3072, ..., 0.0994, 0.7293, 0.3187],
          [0.2723, 0.6290, 0.7742, ..., 0.1179, 0.6817, 0.0761],
          ...,
          [0.2667, 0.3814, 0.9479, ..., 0.4457, 0.3651, 0.0825],
          [0.7468, 0.6061, 0.9087, ..., 0.1951, 0.1604, 0.0321],
          [0.6896, 0.4760, 0.0178, ..., 0.3982, 0.9280, 0.0480]]]]])
MC lowerbound
1: tensor([0.0437, 0.5014], grad_fn=<MeanBackward1>)
2: tensor([0.1282, 0.4484], grad_fn=<MeanBackward1>)

```

4.34.4 3. Learning

We estimate θ using stochastic gradient-based maximum likelihood estimation. For a tractable model, we can assess the log-likelihood function

$$\mathcal{L}(\theta|\mathcal{D}) = \sum_{x \in \mathcal{D}} \log p_X(x|\theta) \quad (4.85)$$

and estimate $\nabla_\theta \mathcal{L}(\theta|\mathcal{D})$ using random mini-batches:

$$\nabla_\theta \mathcal{L}(\theta|\mathcal{D}) \stackrel{\text{MC}}{\approx} \frac{1}{S} \sum_{s=1}^S \nabla_\theta \log p_X(x^{(s)}|\theta) \quad (4.86)$$

$$\text{where } x^{(s)} \sim \mathcal{D} \quad (4.87)$$

An intractable model, such as the continuous LVM above requires approximate inference.

3.1 Intractable LVMs

When marginalisation is intractable, we resort to variational inference introducing a parametric approximation $q_{Z|X=x}$ to the model's true posterior distribution $p_{Z|X=x}$ and estimate both the approximation and the joint distribution by maximising the evidence lowerbound (ELBO), shown below for a single observation x :

$$\mathcal{E}(\lambda, \theta | \mathcal{D}) = \mathbb{E}_{x \sim \mathcal{D}} \left[\mathbb{E} \left[\log \frac{p_{ZX}(z, x | \theta)}{q_{Z|X}(z|x, \lambda)} \right] \right] \quad (4.88)$$

$$= \underbrace{\mathbb{E}_{x \sim \mathcal{D}} [\mathbb{E}[\log p_{Z|X}(x|z, \theta)]]}_{-D} - \underbrace{\mathbb{E}_{x \sim \mathcal{D}} [\text{KL}(q_{Z|X=x} || p_Z)]}_{R} \quad (4.89)$$

where the inner expectation is taken with respect to $q_{Z|X}(z|x, \lambda)$.

When computed in expectation under the data distribution, the two components of the ELBO, namely, the expected log-likelihood $\mathbb{E}[\log p_{Z|X}(x|z, \theta)]$ and the “KL term” $\text{KL}(q_{Z|X=x} || p_Z)$, are related to two information-theoretic quantities known as *distortion* and *rate*.

We choose our approximation to be such that: its support is embedded in the support of the prior, it is simple enough to sample from, and it is simple enough to assess the mass of a sample. If possible, we choose it such that other quantities are also tractable (e.g., entropy, relative entropy).

For a multivariate z , we normally choose a factorised family, for example, if z is a point in \mathbb{R}^K :

$$q_{Z|X}(z|x, \lambda) = \prod_{k=1}^K \mathcal{N}(z_k | \mu_k(x; \lambda), \sigma_k^2(x; \lambda)) \quad (4.90)$$

with $\boldsymbol{\mu}(x; \lambda) \in \mathbb{R}^K$ and $\boldsymbol{\sigma}(x; \lambda) \in \mathbb{R}_{>0}^K$. This is called a *mean field assumption*.

We can obtain a more complex approximation by, for example, using a mixture of mean field families:

$$q_{Z|X}(z|x, \lambda) = \sum_{c=1}^C \omega_c(x; \lambda) \prod_{k=1}^K \mathcal{N}(z_k | \mu_k(x; \lambda), \sigma_k^2(x; \lambda)) \quad (4.91)$$

with $\boldsymbol{\mu}(x; \lambda) \in \mathbb{R}^K$, $\boldsymbol{\sigma}(x; \lambda) \in \mathbb{R}_{>0}^K$, and $\boldsymbol{\omega}(x; \lambda) \in \Delta_{C-1}$.

There are other ways to inject structure in the variational approximation, a common example is to use a normalising flow. When designing a structured approximation a few things must be kept in mind:

- sampling should remain tractable
- assessing the density of a sample should remain tractable
- it's okay if we cannot compute entropy or KL in closed-form, we can always estimate the gradient of such terms (e.g., via score function estimation)

As we shall see the two approximations above differ in a crucial way, the simple Gaussian mean field is amenable to a continuously differentiable reparameterisation which leads to a lower variance estimator (compared to, for example, the score function estimator).

[]:

3.1.1 Reparameterised gradient

For some distributions, it is possible to obtain a sample via a continuously differentiable transformation of a fixed random source. This enables a class for gradient estimators known as *reparameterised gradient* (or the “reparameterisation trick”). In these cases $Z = \mathcal{T}(\epsilon, \lambda)$ with ϵ drawn from a distribution whose parameters are independent of λ . Moreover, \mathcal{T} is differentiable and invertible.

3.2 Inference model

The inference model is a conditional model of the latent variable, for which we design CPD nets.

Before we go on, it is useful to design an “encoder” a function that maps an image $x \in \mathcal{X}$ to a fixed-size vector that we can use as a compact representation of x . Next, we design one such encoder employing FFNNs and another employing CNNs.

```
[27]: class FlattenImage(nn.Module):
    def forward(self, input):
        return input.reshape(input.shape[:-3] + (-1,))

def build_ffnn_encoder(num_channels, width=64, height=64, output_size=1024, p_drop=0.):
    encoder = nn.Sequential(
        FlattenImage(),
        nn.Dropout(p_drop),
        nn.Linear(num_channels * width * height, output_size//2),
        nn.ReLU(),
        nn.Dropout(p_drop),
        nn.Linear(output_size//2, output_size//2),
        nn.ReLU(),
        nn.Dropout(p_drop),
        nn.Linear(output_size//2, output_size),
    )
    return encoder

def build_cnn_encoder(num_channels, width=64, height=64, output_size=1024, p_drop=0.):
    if width != 64:
        raise ValueError("The width is hardcoded")
    if height != 64:
        raise ValueError("The height is hardcoded")
    if output_size != 1024:
        raise ValueError("The output_size is hardcoded")
    # TODO: change the architecture so width, height and output_size are not hardcoded
    encoder = MySequential(
        nn.Conv2d(num_channels, 32, 4, 2),
        nn.LeakyReLU(0.2),
        nn.Conv2d(32, 64, 4, 2),
        nn.LeakyReLU(0.2),
        nn.Conv2d(64, 128, 4, 2),
        nn.LeakyReLU(0.2),
        nn.Conv2d(128, 256, 4, 2),
        nn.LeakyReLU(0.2),
        FlattenImage(),
        event_dims=3
    )
```

(continues on next page)

(continued from previous page)

```
)  
    return encoder
```

[28]: # a batch of five [1, 64, 64]-dimensional images is encoded into
five 1024-dimensional vectors
build_ffnn_encoder(num_channels=1)(torch.zeros((5, 1, 64, 64))).shape
[28]: torch.Size([5, 1024])

[29]: # and, again, we can have structured batches
(here trying with (3,5))
build_ffnn_encoder(num_channels=1)(torch.zeros((3, 5, 1, 64, 64))).shape
[29]: torch.Size([3, 5, 1024])

[30]: # a batch of five [1, 64, 64]-dimensional images is encoded into
five 1024-dimensional vectors
build_cnn_encoder(num_channels=1)(torch.zeros((5, 1, 64, 64))).shape
[30]: torch.Size([5, 1024])

[31]: # and, again, since we use MySequential we can have structured batches
(here trying with (3,5))
build_cnn_encoder(num_channels=1)(torch.zeros((3, 5, 1, 64, 64))).shape
[31]: torch.Size([3, 5, 1024])

We can now design some CPD nets, assuming they map from an encoding of an image to a pmf over \mathcal{Z} .

Gaussian mean field

This can be used to parameterise a cpd over real vectors of fixed dimensionality.

Mixture of Gaussian mean fields

This can also be used to parameterise a cpd over real vectors of fixed dimensionality, but it achieves a more complex density (e.g., multimodal).

[32]: `class GaussianCPDNet(CPDNet):
 """
 Output distribution is a product of Gaussian distributions
 """

 def __init__(self, outcome_shape, num_inputs: int, hidden_size: int=None, p_drop:_
 -float=0.):
 """
 outcome_shape: shape of the outcome (int or tuple)
 if int, we turn it into a singleton tuple
 num_inputs: rightmost dimensionality of the inputs to forward
 hidden_size: size of hidden layers for the CPDNet (use None to skip)
 p_drop: configure dropout before every Linear layer
 """
 super().__init__(outcome_shape)
 num_outputs = np.prod(self.outcome_shape)`

(continues on next page)

(continued from previous page)

```

if hidden_size:
    self.encoder = nn.Sequential(
        nn.Dropout(p_drop),
        nn.Linear(num_inputs, hidden_size),
        nn.ReLU()
    )
else:
    self.encoder = nn.Identity()
    hidden_size = num_inputs

self.locs = nn.Sequential(
    nn.Dropout(p_drop),
    nn.Linear(hidden_size, num_outputs),
    ReshapeLast(self.outcome_shape)
)
self.scales = nn.Sequential(
    nn.Dropout(p_drop),
    nn.Linear(hidden_size, num_outputs),
    nn.Softplus(), # we use the softplus activations for the scales
    ReshapeLast(self.outcome_shape)
)

def forward(self, inputs):
    h = self.encoder(inputs)
    return td.Independent(td.Normal(loc=self.locs(h), scale=self.scales(h)), ↴
    len(self.outcome_shape))

class MoGCPDNet(CPDNet):
    """
    Output distribution is a mixture of products of Gaussian distributions
    """

    def __init__(self, outcome_shape, num_inputs: int, hidden_size: int=None, p_drop: float=0., num_components=2):
        """
        outcome_shape: shape of the outcome (int or tuple)
        if int, we turn it into a singleton tuple
        num_inputs: rightmost dimensionality of the inputs to forward
        hidden_size: size of hidden layers for the CPDNet (use None to skip)
        p_drop: configure dropout before every Linear layer
        num_components: number of Gaussians to be mixed
        """
        super().__init__(outcome_shape)
        self.num_components = num_components

        num_outputs = num_components * np.prod(self.outcome_shape)

        if hidden_size:
            self.encoder = nn.Sequential(
                nn.Dropout(p_drop),
                nn.Linear(num_inputs, hidden_size),
                nn.ReLU()
            )

```

(continues on next page)

(continued from previous page)

```

        )
    else:
        self.encoder = nn.Identity()
        hidden_size = num_inputs

    self.locs = nn.Sequential(
        nn.Dropout(p_drop),
        nn.Linear(hidden_size, num_outputs),
        ReshapeLast((num_components,) + self.outcome_shape)
    )
    self.scales = nn.Sequential(
        nn.Dropout(p_drop),
        nn.Linear(hidden_size, num_outputs),
        nn.Softplus(), # we use the softplus activations for the scales
        ReshapeLast((num_components,) + self.outcome_shape)
    )
    self.logits = nn.Sequential(
        nn.Dropout(p_drop),
        nn.Linear(hidden_size, num_components),
        ReshapeLast((num_components,))
)

def forward(self, inputs):
    h = self.encoder(inputs)
    comps = td.Independent(td.Normal(loc=self.locs(h), scale=self.scales(h)), ↴
    len(self.outcome_shape))
    pc = td.Categorical(logits=self.logits(h))
    return td.MixtureSameFamily(pc, comps)

```

[33]: def test_cpds(outcome_shape, num_comps=2, batch_size=3, input_dim=5, hidden_size=2):

```

cpd_net = GaussianCPDNet(outcome_shape, num_inputs=input_dim, hidden_size=hidden_
size)
print("\nGaussian")
print(cpd_net)
print(f" outcome_shape={cpd_net.outcome_shape}")
inputs = torch.from_numpy(np.random.uniform(size=(batch_size, input_dim))).float()
print(f" shape of inputs: {inputs.shape}")
p = cpd_net(inputs)
print(f" distribution: {p}")
z = p.sample()
print(f" sample: {z}")
print(f" shapes: sample={z.shape} log_prob={p.log_prob(z).shape}")

cpd_net = MoGCPDNet(outcome_shape, num_inputs=input_dim, hidden_size=hidden_size, ↴
num_components=num_comps)
print("\nMixture of Gaussians")
print(cpd_net)
print(f" outcome_shape={cpd_net.outcome_shape}")
inputs = torch.from_numpy(np.random.uniform(size=(batch_size, input_dim))).float()
print(f" shape of inputs: {inputs.shape}")
p = cpd_net(inputs)

```

(continues on next page)

(continued from previous page)

```

print(f" distribution: {p}")
z = p.sample()
print(f" sample: {z}")
print(f" shapes: sample={z.shape} log_prob={p.log_prob(z).shape}")

# Try a few
test_cpds(12)
#test_cpds(12, hidden_size=None)
# your latent code could be a matrix (we talk about it as a "vector" for convenience)
#test_cpds((4, 5))

Gaussian
GaussianCPDNet(
    encoder: Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=5, out_features=2, bias=True)
        (2): ReLU()
    )
    (locs): Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=2, out_features=12, bias=True)
        (2): ReshapeLast()
    )
    (scales): Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=2, out_features=12, bias=True)
        (2): Softplus(beta=1, threshold=20)
        (3): ReshapeLast()
    )
)
outcome_shape=(12,)
shape of inputs: torch.Size([3, 5])
distribution: Independent(Normal(loc: torch.Size([3, 12]), scale: torch.Size([3, 12])), -1)
sample: tensor([-0.2740,  1.4641, -0.1492,  0.8619,  0.8987, -0.6384,  0.1898, -0.0084,
               0.0154, -0.0720,  0.5614, -0.7824],
              [ 1.4560, -0.2668, -0.2343, -0.8925,  0.0078, -0.1212,  0.3651, -0.8519,
               1.0650, -1.2128, -0.4736, -0.1115],
              [ 0.1504, -0.5747,  0.1681,  0.4921,  1.2217, -0.4474, -0.9136, -1.1818,
               0.9822, -0.6862, -0.0885, -0.3536]])
shapes: sample=torch.Size([3, 12]) log_prob=torch.Size([3])

Mixture of Gaussians
MoGCPDNet(
    encoder: Sequential(
        (0): Dropout(p=0.0, inplace=False)
        (1): Linear(in_features=5, out_features=2, bias=True)
        (2): ReLU()
    )
    (locs): Sequential(

```

(continues on next page)

(continued from previous page)

```
(0): Dropout(p=0.0, inplace=False)
(1): Linear(in_features=2, out_features=24, bias=True)
(2): ReshapeLast()
)
(scales): Sequential(
(0): Dropout(p=0.0, inplace=False)
(1): Linear(in_features=2, out_features=24, bias=True)
(2): Softplus(beta=1, threshold=20)
(3): ReshapeLast()
)
(logits): Sequential(
(0): Dropout(p=0.0, inplace=False)
(1): Linear(in_features=2, out_features=2, bias=True)
(2): ReshapeLast()
)
)
)
outcome_shape=(12,)
shape of inputs: torch.Size([3, 5])
distribution: MixtureSameFamily(
  Categorical(logits: torch.Size([3, 2])),
  Independent(Normal(loc: torch.Size([3, 2, 12])), scale: torch.Size([3, 2, 12])), 1))
sample: tensor([[ 2.2989,  0.6416, -0.7611, -0.2090, -0.4584,  0.5179, -0.1198,  0.0880,
    0.7712, -0.4315, -0.3756,  0.2588],
   [-0.1332, -0.0895, -0.0302,  1.7607, -0.6219,  0.5210,  1.3052,  0.4076,
    0.2439, -0.4406,  0.0756,  0.6263],
   [-0.1179,  1.3194, -0.2864,  1.0093,  1.1053,  0.9005,  0.6088,  0.2438,
    0.7631, -0.8659,  0.4035,  0.1295]])
shapes: sample=torch.Size([3, 12]) log_prob=torch.Size([3])
```

Last, but certainly not least, we can combine our encoder and a choice of CPD net.

```
[34]: class InferenceModel(CPDNet):

    def __init__(
        self, cpd_net_type,
        latent_size, num_channels=1, width=64, height=64,
        hidden_size=1024, p_drop=0.,
        encoder_type=build_ffnn_encoder):

        super().__init__(latent_size)

        self.latent_size = latent_size
        # encodes an image to a hidden_size-dimensional vector
        self.encoder = encoder_type(
            num_channels=num_channels,
            width=width,
            height=height,
            output_size=hidden_size,
            p_drop=p_drop
        )
        # maps from a hidden_size-dimensional encoding
        # to a cpd for Z/X=x
```

(continues on next page)

(continued from previous page)

```

self.cpd_net = cpd_net_type(
    latent_size,
    num_inputs=hidden_size,
    hidden_size=2*latent_size,
    p_drop=p_drop
)

def forward(self, x):
    h = self.encoder(x)
    return self.cpd_net(h)

```

[35]: InferenceModel(GaussianCPDNet, latent_size=10)(torch.zeros(5, 1, 64, 64))

[35]: Independent(Normal(loc: torch.Size([5, 10]), scale: torch.Size([5, 10])), 1)

[36]: InferenceModel(partial(MoGCPDNet, num_components=3), latent_size=10)(torch.zeros(5, 1, 64, 64))

[36]: MixtureSameFamily(
 Categorical(logits: torch.Size([5, 3])),
 Independent(Normal(loc: torch.Size([5, 3, 10]), scale: torch.Size([5, 3, 10])), 1))

We now have everything in place to use variational inference.

3.3 Neural Variational Inference

We will train our generative model via variational inference, for which we need to train an inference model along with it. We will use the ELBO objective, and gradient estimators based on score function estimation and differentiable reparameterisation.

It's common to refer to any one such model as a variational auto-encoder (VAE), especially so when using reparameterised gradients.

Let's start with score function estimation.

Given a data point x , we estimate the gradient of the ELBO with respect to λ by MC estimating the following expressions:

$$\nabla_\lambda \mathcal{E}(\lambda, \theta|x) = \mathbb{E} \left[r(z, x; \theta, \lambda) \nabla_\lambda \log \frac{p_{ZX}(z, x|\theta)}{q_{Z|X}(z|x, \lambda)} \right] \quad (4.92)$$

$$\nabla_\theta \mathcal{E}(\lambda, \theta|x) = \mathbb{E} [\nabla_\theta \log p_{ZX}(z, x|\theta)] \quad (4.93)$$

where the “reward” function in the gradient estimator for λ is

$$r(z, x; \theta, \lambda) = \log p_{X|Z}(x|z, \theta) \quad (4.94)$$

And, because this gradient estimator is rather noisy, it's common to transform the reward function by further employing control variates. The simplest control variates are functions of x and possibly of θ and λ , but not a function of the action z with respect to which we evaluate the reward function. We will implement those as wrappers around the reward function. So, let's start by agreeing on the API of our control variates.

```
[37]: class VarianceReduction(nn.Module):
    """
    We will be using simple forms of control variates for variance reduction.
    These are transformations of the reward that are independent of the sampled
    latent variable, but they can, in principle, depend on x, and on the
    parameters of the generative and inference model.

    Some of these are trainable components, thus they also contribute to the loss.
    """

    def __init__(self):
        super().__init__()

    def forward(self, r, x, q, r_fn):
        """
        Return the transformed reward and a contribution to the loss.

        r: a batch of rewards
        x: a batch of observations
        q: policy
        r_fn: reward function
        """
        return r, torch.zeros_like(r)
```

In case a reparameterisation trick is available for the approximate posterior, we can MC estimate the following expressions:

$$\nabla_{\lambda} \mathcal{E}(\lambda, \theta|x) = \mathbb{E}_{\epsilon \sim s(\cdot)} \left[\nabla_{\lambda} \log \frac{p_{XZ}(x, Z = \mathcal{T}(\epsilon, \lambda)|\theta)}{q_{Z|X}(Z = \mathcal{T}(\epsilon, \lambda)|x, \lambda)} \right] \quad (4.95)$$

and

$$\nabla_{\theta} \mathcal{E}(\lambda, \theta|x) = \mathbb{E}_{\epsilon \sim s(\cdot)} \left[\nabla_{\theta} \log \frac{p_{XZ}(x, Z = \mathcal{T}(\epsilon, \lambda)|\theta)}{q_{Z|X}(Z = \mathcal{T}(\epsilon, \lambda)|x, \lambda)} \right] \quad (4.96)$$

Gradients of these kind are commonly referred to as *path derivatives*. We don't need to implement the path derivatives ourselves, nor the transformations, rather we use a distribution object which supports an `rsample` method (for “reparameterised sample”), this distribution will be able to assess the density of the sample should we need it and if we obtained the sample via `rsample` the path derivative will be automatically available to backprop.

If KL divergence from the prior to the approximate posterior is computable, we use a different gradient estimator for λ and θ , namely:

$$\nabla_{\lambda} \mathcal{E}(\lambda, \theta|x) = \mathbb{E}_{\epsilon \sim s(\cdot)} [\nabla_{\lambda} \log p_{X|Z}(x|Z = \mathcal{T}(\epsilon, \lambda), \theta)] - \nabla_{\lambda} \text{KL}(q_{Z|X}||p_Z) \quad (4.97)$$

and

$$\nabla_{\theta} \mathcal{E}(\lambda, \theta|x) = \mathbb{E}_{\epsilon \sim s(\cdot)} [\nabla_{\theta} \log p_{X|Z}(x|Z = \mathcal{T}(\epsilon, \lambda), \theta)] - \nabla_{\theta} \text{KL}(q_{Z|X}||p_Z) \quad (4.98)$$

Recall that, in practice, we will need to design a surrogate loss: a node in the computation graph whose backward corresponds to the gradient estimator we want. Check carefully the changes we made to the forward method of the NVIL class.

Now we can work on our general NVIL model. The following class implements the NVIL objective as well as a lot of helper code to manipulate the model components in interesting ways (e.g., sampling, sampling conditionally, estimating marginal density, etc.)

```
[38]: class NVIL(nn.Module):
    """
    A generative model  $p(z)p(x|z)$  and an approximation  $q(z|x)$  to that
    model's true posterior.

    The approximation is estimated to maximise the ELBO, and so is the joint
    distribution.
    """

    def __init__(self, gen_model: JointDistribution, inf_model: InferenceModel, cv_model:
        ↪ VarianceReduction):
        """
        gen_model:  $p(z)p(x|z)$ 
        inf_model:  $q(z|x)$  which approximates  $p(z|x)$ 
        cv_model: optional transformations of the reward
        """

        super().__init__()
        self.gen_model = gen_model
        self.inf_model = inf_model
        self.cv_model = cv_model

    def gen_params(self):
        return self.gen_model.parameters()

    def inf_params(self):
        return self.inf_model.parameters()

    def cv_params(self):
        return self.cv_model.parameters()

    def sample(self, batch_size, sample_size=None, oversample=False):
        """
        A sample from the joint distribution:
        z ~ prior
        x/z ~ obs model
        batch_size: number of samples in a batch
        sample_size: if None, the output tensor has shape [batch_size] + data_shape
                     if 1 or more, the output tensor has shape [sample_size, batch_size] + data_
        ↪ shape
                     while batch_size controls a parallel computation,
                     sample_size controls a sequential computation (a for loop)
        oversample: if True, samples z (batch_size times), hold it fixed,
                    and sample x (sample_size times)
        """

        pz = self.gen_model.prior((batch_size,))
        samples = [None] * (sample_size or 1)
        px_z = self.gen_model.obs_model(pz.sample()) if oversample else None
        for k in range(sample_size or 1):
            if not oversample:
                px_z = self.gen_model.obs_model(pz.sample())
            samples[k] = px_z.sample()
        x = torch.stack(samples)
        return x if sample_size else x.squeeze(0)
```

(continues on next page)

(continued from previous page)

```

def cond_sample(self, x, sample_size=None, oversample=False):
    """
    Condition on x and draw a sample:
        z/x ~ inf model
        x'/z ~ obs model

    x: a batch of seed data samples
    sample_size: if None, the output tensor has shape [batch_size] + data_shape
                 if 1 or more, the output tensor has shape [sample_size, batch_size] + data_
shape
        sample_size controls a sequential computation (a for loop)
    oversample: if True, samples z (batch_size times), hold it fixed,
                and sample x' (sample_size times)
    """
    qz = self.inf_model(x)
    samples = [None] * (sample_size or 1)
    px_z = self.gen_model.obs_model(qz.sample()) if oversample else None
    for k in range(sample_size or 1):
        if not oversample:
            px_z = self.gen_model.obs_model(qz.sample())
            samples[k] = px_z.sample()
    x = torch.stack(samples)
    return x if sample_size else x.squeeze(0)

def log_prob(self, z, x):
    """
    The log density of the joint outcome under the generative model
    z: [batch_size, latent_dim]
    x: [batch_size] + data_shape
    """
    return self.gen_model.log_prob(z=z, x=x)

def DRL(self, x, sample_size=None):
    """
    MC estimates of a model's
        * distortion D
        * rate R
        * and log-likelihood L
    The estimates are based on single data points
    but multiple latent samples.

    x: batch_shape + data_shape
    sample_size: if 1 or more, we use multiple samples
                 sample_size controls a sequential computation (a for loop)
    """
    sample_size = sample_size or 1
    obs_dims = len(self.gen_model.cpd_net.outcome_shape)
    batch_shape = x.shape[:-obs_dims]
    with torch.no_grad():
        qz = self.inf_model(x)
        pz = self.gen_model.prior(batch_shape)

```

(continues on next page)

(continued from previous page)

```

try: # not every design admits tractable KL
    R = td.kl_divergence(qz, pz)
except NotImplementedError:
    # MC estimation of  $KL(q(z|x) || p(z))$ 
    z = qz.sample((sample_size,))
    R = (qz.log_prob(z) - pz.log_prob(z)).mean(0)
D = 0
ratios = [None] * sample_size
for k in range(sample_size):
    z = qz.sample()
    px_z = self.gen_model.obs_model(z)
    ratios[k] = pz.log_prob(z) + px_z.log_prob(x) - qz.log_prob(z)
    D = D - px_z.log_prob(x)
ratios = torch.stack(ratios, dim=-1)
L = torch.logsumexp(ratios, dim=-1) - np.log(sample_size)
D = D / sample_size
return D, R, L

def elbo(self, x, sample_size=None):
    """
    An MC estimate of ELBO = -D -R

    x: [batch_size] + data_shape
    sample_size: if 1 or more, we use multiple samples
        sample_size controls a sequential computation (a for loop)
    """
    D, R, _ = self.DRL(x, sample_size=sample_size)
    return -D -R

def log_prob_estimate(self, x, sample_size=None):
    """
    An importance sampling estimate of log p(x)

    x: [batch_size] + data_shape
    sample_size: if 1 or more, we use multiple samples
        sample_size controls a sequential computation (a for loop)
    """
    _, _, L = self.DRL(x, sample_size=sample_size)
    return L

def forward(self, x, sample_size=None, rate_weight=1.):
    """
    A surrogate for an MC estimate of - grad ELBO

    x: [batch_size] + data_shape
    sample_size: if 1 or more, we use multiple samples
        sample_size controls a sequential computation (a for loop)
    cv: optional module for variance reduction
    """
    sample_size = sample_size or 1
    obs_dims = len(self.gen_model.cpd_net.outcome_shape)
    batch_shape = x.shape[:-obs_dims]

```

(continues on next page)

(continued from previous page)

```

qz = self.inf_model(x)
pz = self.gen_model.prior(batch_shape)

# we can *always* make use of the score function estimator (SFE)
use_sfe = True

# these 3 log densities will contribute to the different parts of the objective
log_p_x_z = 0.
log_p_z = 0.
log_q_z_x = 0.

# these quantities will help us compute the SFE part of the objective
# (if needed)
sfe = 0
reward = 0
cv_reward = 0
raw_r = 0
cv_loss = 0

for _ in range(sample_size):

    # Obtain a sample
    if qz.has_rsample: # this is how td objects tell us whether they are
continuously reparameterisable
        z = qz.rsample()
        use_sfe = False # with path derivatives, we do not need SFE
    else:
        z = qz.sample()

    # Parameterise the observational model
    px_z = self.gen_model.obs_model(z)

    # Compute all three relevant densities:
    # p(x|z,theta)
    log_p_x_z = log_p_x_z + px_z.log_prob(x)
    # q(z|x,lambda)
    log_q_z_x = log_q_z_x + qz.log_prob(z)
    # p(z|theta)
    log_p_z = log_p_z + pz.log_prob(z)

    # Compute the "reward" for SFE
    raw_r = log_p_x_z + log_p_z - log_q_z_x

    # Apply variance reduction techniques
    r, l = self.cv_model(raw_r.detach(), x=x, q=qz, r_fn=lambda a: self.gen_
model(a).log_prob(x))
    cv_loss = cv_loss + l

    # SFE part for updating lambda
    sfe = sfe + r.detach() * qz.log_prob(z)

```

(continues on next page)

(continued from previous page)

```

# Compute the sample mean for the different terms
sfe = (sfe / sample_size)
cv_loss = cv_loss / sample_size
log_p_x_z = log_p_x_z / sample_size
log_p_z = log_p_z / sample_size
log_q_z_x = log_q_z_x / sample_size

D = - log_p_x_z
try: # not every design admits tractable KL
    R = td.kl_divergence(qz, pz)
except NotImplementedError:
    R = log_q_z_x - log_p_z

if use_sfe:
    # the first two terms update theta
    # the last term updates lambda
    elbo_grad_surrogate = log_p_x_z + log_p_z + sfe
    # note that the term (log_p_x_z + log_p_z) is also part of sfe
    # but there it is detached, meaning that it won't contribute to
    # grad theta
else:
    # without SFE, we can use the classic form of the ELBO
    elbo_grad_surrogate = -D - R

loss = -elbo_grad_surrogate + cv_loss

return {'loss': loss.mean(0), 'ELBO': (-D -R).mean(0).item(), 'D': D.mean(0).item(),
        'R': R.mean(0).item(), 'cv_loss': cv_loss.mean(0).item()}

```

Here's an example

```
[39]: vae = NVIL(
    JointDistribution(
        GaussianPriorNet(10),
        BinarizedImageModel(
            num_channels=img_shape[0],
            width=img_shape[1],
            height=img_shape[2],
            latent_size=10,
            p_drop=0.1,
            decoder_type=build_ffnn_decoder
        )
    ),
    InferenceModel(
        cpd_net_type=GaussianCPDNet,
        latent_size=10,
        num_channels=img_shape[0],
        width=img_shape[1],
        height=img_shape[2],
        encoder_type=build_ffnn_encoder
    ),
    VarianceReduction()
)
```

(continues on next page)

(continued from previous page)

```
)  
vae  
[39]: NVIL(  
    (gen_model): JointDistribution(  
        (prior_net): GaussianPriorNet()  
        (cpd_net): BinarizedImageModel(  
            (decoder): Sequential(  
                (0): Dropout(p=0.1, inplace=False)  
                (1): Linear(in_features=10, out_features=512, bias=True)  
                (2): ReLU()  
                (3): Dropout(p=0.1, inplace=False)  
                (4): Linear(in_features=512, out_features=512, bias=True)  
                (5): ReLU()  
                (6): Dropout(p=0.1, inplace=False)  
                (7): Linear(in_features=512, out_features=4096, bias=True)  
                (8): ReshapeLast()  
            )  
        )  
    )  
    (inf_model): InferenceModel(  
        (encoder): Sequential(  
            (0): FlattenImage()  
            (1): Dropout(p=0.0, inplace=False)  
            (2): Linear(in_features=4096, out_features=512, bias=True)  
            (3): ReLU()  
            (4): Dropout(p=0.0, inplace=False)  
            (5): Linear(in_features=512, out_features=512, bias=True)  
            (6): ReLU()  
            (7): Dropout(p=0.0, inplace=False)  
            (8): Linear(in_features=512, out_features=1024, bias=True)  
        )  
        (cpd_net): GaussianCPDNet(  
            (encoder): Sequential(  
                (0): Dropout(p=0.0, inplace=False)  
                (1): Linear(in_features=1024, out_features=20, bias=True)  
                (2): ReLU()  
            )  
            (locs): Sequential(  
                (0): Dropout(p=0.0, inplace=False)  
                (1): Linear(in_features=20, out_features=10, bias=True)  
                (2): ReshapeLast()  
            )  
            (scales): Sequential(  
                (0): Dropout(p=0.0, inplace=False)  
                (1): Linear(in_features=20, out_features=10, bias=True)  
                (2): Softplus(beta=1, threshold=20)  
                (3): ReshapeLast()  
            )  
        )  
        (cv_model): VarianceReduction()  
)
```

```
[40]: for x, y in train_loader:
    print('x.shape:', x.shape)
    print(vae(x))
    break

x.shape: torch.Size([64, 1, 64, 64])
{'loss': tensor(2845.1970, grad_fn=<MeanBackward1>), 'ELBO': -2845.197021484375, 'D': -2843.9189453125, 'R': 1.27774977684021, 'cv_loss': 0.0}
```

3.3.1 Training algorithm

We have up to three components (recall that some control variates can have their own parameters), so we will be manipulating up to three optimisers:

```
[41]: class OptCollection:

    def __init__(self, gen, inf, cv=None):
        self.gen = gen
        self.inf = inf
        self.cv = cv

    def zero_grad(self):
        self.gen.zero_grad()
        self.inf.zero_grad()
        if self.cv:
            self.cv.zero_grad()

    def step(self):
        self.gen.step()
        self.inf.step()
        if self.cv:
            self.cv.step()
```

Here's some helper code to assess and train the model

```
[42]: from collections import defaultdict, OrderedDict
from tqdm.auto import tqdm

def assess(model, sample_size, dl, device):
    """
    Wrapper for estimating a model's ELBO, distortion, rate, and log-likelihood
    using all data points in a data loader.
    """
    D = 0
    R = 0
    L = 0
    data_size = 0
    with torch.no_grad():
        for batch_x, batch_y in dl:
            Dx, Rx, Lx = model.DRL(batch_x.to(device), sample_size=sample_size)
            D = D + Dx.sum(0)
```

(continues on next page)

(continued from previous page)

```

R = R + Rx.sum(0)
L = L + Lx.sum(0)
data_size += batch_x.shape[0]
D = D / data_size
R = R / data_size
L = L / data_size
return {'ELBO': (-D -R).item(), 'D': D.item(), 'R': R.item(), 'L': L.item()}

def train_vae(model: NVIL, opts: OptCollection,
    training_data, dev_data,
    batch_size=64, num_epochs=10, check_every=10,
    sample_size_training=1,
    sample_size_eval=10,
    grad_clip=5.,
    num_workers=2,
    device=torch.device('cuda:0'))
):
    """
    model: pytorch model
    optimiser: pytorch optimiser
    training_corpus: a TaggedCorpus for trianing
    dev_corpus: a TaggedCorpus for dev
    batch_size: use more if you have more memory
    num_epochs: use more for improved convergence
    check_every: use less to check performance on dev set more often
    device: where we run the experiment

    Return a log of quantities computed during training (for plotting)
    """
    batcher = DataLoader(training_data, batch_size, shuffle=True, num_workers=num_workers, pin_memory=True)
    dev_batcher = DataLoader(dev_data, batch_size, num_workers=num_workers, pin_memory=True)

    total_steps = num_epochs * len(batcher)
    log = defaultdict(list)

    step = 0
    model.eval()
    for k, v in assess(model, sample_size_eval, dev_batcher, device=device).items():
        log[f"dev.{k}"].append((step, v))

    with tqdm(range(total_steps)) as bar:
        for epoch in range(num_epochs):
            for batch_x, batch_y in batcher:
                model.train()
                opts.zero_grad()

                loss_dict = model(
                    batch_x.to(device),
                    sample_size=sample_size_training,

```

(continues on next page)

(continued from previous page)

```

        )
    for metric, value in loss_dict.items():
        log[f'training.{metric}'].append((step, value))

    loss_dict['loss'].backward()

    nn.utils.clip_grad_norm_(
        model.parameters(),
        grad_clip
    )
    opts.step()

    bar_dict = OrderedDict()
    for metric, value in loss_dict.items():
        bar_dict[f'training.{metric}'] = f'{loss_dict[metric]:.2f}'
    for metric in ['ELBO', 'D', 'R', 'L']:
        bar_dict[f'dev.{metric}'] = f'{:.2f}'.format(log[f'dev.{metric}'][-1][1])
    bar.set_postfix(bar_dict)
    bar.update()

    if step % check_every == 0:
        model.eval()
        for k, v in assess(model, sample_size_eval, dev_batcher, device=device).items():
            log[f'dev.{k}'].append((step, v))

    step += 1

model.eval()
for k, v in assess(model, sample_size_eval, dev_batcher, device=device).items():
    log[f'dev.{k}'].append((step, v))

return log

```

And, finally, some code to help inspect samples

```
[43]: def inspect_lvm(model, dl, device):
    for x, y in dl:

        x_ = model.sample(16, 4, oversample=True).cpu().reshape(-1, 1, 64, 64)
        plt.figure(figsize=(16,8))
        plt.axis('off')
        plt.imshow(make_grid(x_, nrow=16).permute((1, 2, 0)))
        plt.title("Prior samples")
        plt.show()

        plt.figure(figsize=(16,8))
        plt.axis('off')
        plt.imshow(make_grid(x, nrow=16).permute((1, 2, 0)))
        plt.title("Observations")
```

(continues on next page)

(continued from previous page)

```

plt.show()

x_ = model.cond_sample(x.to(device)).cpu().reshape(-1, 1, 64, 64)
plt.figure(figsize=(16,8))
plt.axis('off')
plt.imshow(make_grid(x_, nrow=16).permute((1, 2, 0)))
plt.title("Conditional samples")
plt.show()

break

```

3.3.2 Variance reduction

Here are some concrete strategies for variance reduction. You can skip those in a first pass.

```
[44]: class CentredReward(VarianceReduction):
    """
    This control variate does not have trainable parameters,
    it maintains a running estimate of the average reward and updates
    a batch of rewards by computing reward - avg.
    """

    def __init__(self, alpha=0.9):
        super().__init__()
        self._alpha = alpha
        self._r_mean = 0.

    def forward(self, r, x=None, q=None, r_fn=None):
        """
        Centre the reward and update running estimates of mean.
        """
        with torch.no_grad():
            # sufficient statistics for next updates
            r_mean = torch.mean(r, dim=0)
            # centre the signal
            r = r - self._r_mean
            # update running estimate of mean
            self._r_mean = (1-self._alpha) * self._r_mean + self._alpha * r_mean.item()
        return r, torch.zeros_like(r)

class ScaledReward(VarianceReduction):
    """
    This control variate does not have trainable parameters,
    it maintains a running estimate of the reward's standard deviation and
    updates a batch of rewards by computing reward / maximum(stddev, 1).
    """

    def __init__(self, alpha=0.9):
        super().__init__()
        self._alpha = alpha
```

(continues on next page)

(continued from previous page)

```

self._r_std = 1.0

def forward(self, r, x=None, q=None, r_fn=None):
    """
    Scale the reward by a running estimate of std, and also update the estimate.
    """
    with torch.no_grad():
        # sufficient statistics for next updates
        r_std = torch.std(r, dim=0)
        # standardise the signal
        r = r / self._r_std
        # update running estimate of std
        self._r_std = (1-self._alpha) * self._r_std + self._alpha * r_std.item()
        # it's not safe to standardise with scales less than 1
        self._r_std = np.maximum(self._r_std, 1.)
    return r, torch.zeros_like(r)

class SelfCritic(VarianceReduction):
    """
    This control variate does not have trainable parameters,
    it updates a batch of rewards by computing reward - 'reward', where
    'reward' is  $\log p(X=x|Z=z')$ .detach() assessed for a novel sample
     $z' \sim Z|X=x$ .
    """

    def __init__(self):
        super().__init__()

    def forward(self, r, x, q, r_fn):
        """
        Standardise the reward and update running estimates of mean/std.
        """
        with torch.no_grad():
            z = q.sample()
            r = r - r_fn(z, x)
        return r, torch.zeros_like(r)

class Baseline(VarianceReduction):
    """
    An input-dependent baseline implemented as an MLP.
    The trainable parameters are adjusted via MSE.
    """

    def __init__(self, num_inputs, hidden_size, p_drop=0.):
        super().__init__()
        self.baseline = nn.Sequential(
            FlattenImage(),
            nn.Dropout(p_drop),
            nn.Linear(num_inputs, hidden_size),
            nn.ReLU(),

```

(continues on next page)

(continued from previous page)

```

        nn.Dropout(p_drop),
        nn.Linear(hidden_size, 1)
    )

def forward(self, r, x, q=None, r_fn=None):
    """
    Return r - baseline(x) and Baseline's loss.
    """
    # batch_shape + (1,)
    r_hat = self.baseline(x)
    # batch_shape
    r_hat = r_hat.squeeze(-1)
    loss = (r - r_hat)**2
    return r - r_hat.detach(), loss

class CVChain(VarianceReduction):

    def __init__(self, *args):
        super().__init__()
        if len(args) == 1 and isinstance(args[0], OrderedDict):
            for key, module in args[0].items():
                self.add_module(key, module)
        else:
            for idx, module in enumerate(args):
                self.add_module(str(idx), module)

    def forward(self, r, x, q, r_fn):
        loss = 0
        for cv in self._modules.values():
            r, l = cv(r, x=x, q=q, r_fn=r_fn)
            loss = loss + l
        return r, loss

```

3.3.3 Experiment

```
[45]: seed_all()
my_device = torch.device('cuda:0')

model = NVIL(
    JointDistribution(
        GaussianPriorNet(10),
        BinarizedImageModel(
            num_channels=img_shape[0],
            width=img_shape[1],
            height=img_shape[2],
            latent_size=10,
            p_drop=0.1
        )
    ),

```

(continues on next page)

(continued from previous page)

```

InferenceModel(
    latent_size=10,
    num_channels=img_shape[0],
    width=img_shape[1],
    height=img_shape[2],
    cpd_net_type=GaussianCPDNet # Gaussian prior and Gaussian posterior: this is a
˓→classic VAE
),
VarianceReduction(), # no variance reduction is needed for a VAE
#CVChain( # variance reduction helps SFE
#    CentredReward(),
#    #Baseline(np.prod(img_shape), 512), # this is how you would use a trained
˓→baseline
#    #ScaledReward()
#)
).to(my_device)

opts = OptCollection(
    # Tips based on empirical practice:

    # Adam is the go-to choice for (reparameterised) VAEs
    opt.Adam(model.gen_params(), lr=5e-4, weight_decay=1e-6),
    opt.Adam(model.inf_params(), lr=1e-4),

    # Adam is not often a good choice for SFE-based optimisation
    # a possible reason: SFE is too noisy and the design choices behind Adam
    # were made having reparameterised gradients in mind
    #opt.RMSprop(model.gen_params(), lr=5e-4, weight_decay=1e-6),
    #opt.RMSprop(model.inf_params(), lr=1e-4),
    #opt.RMSprop(model.cv_params(), lr=1e-4, weight_decay=1e-6) # you need this if your
˓→baseline has trainable parameters
)

model

```

[45]: NVIL

```

(gen_model): JointDistribution(
    (prior_net): GaussianPriorNet()
    (cpd_net): BinarizedImageModel(
        (decoder): Sequential(
            (0): Dropout(p=0.1, inplace=False)
            (1): Linear(in_features=10, out_features=512, bias=True)
            (2): ReLU()
            (3): Dropout(p=0.1, inplace=False)
            (4): Linear(in_features=512, out_features=512, bias=True)
            (5): ReLU()
            (6): Dropout(p=0.1, inplace=False)
            (7): Linear(in_features=512, out_features=4096, bias=True)
            (8): ReshapeLast()
        )
    )
)

```

(continues on next page)

(continued from previous page)

```

)
(inf_model): InferenceModel(
    (encoder): Sequential(
        (0): FlattenImage()
        (1): Dropout(p=0.0, inplace=False)
        (2): Linear(in_features=4096, out_features=512, bias=True)
        (3): ReLU()
        (4): Dropout(p=0.0, inplace=False)
        (5): Linear(in_features=512, out_features=512, bias=True)
        (6): ReLU()
        (7): Dropout(p=0.0, inplace=False)
        (8): Linear(in_features=512, out_features=1024, bias=True)
    )
    (cpd_net): GaussianCPDNet(
        (encoder): Sequential(
            (0): Dropout(p=0.0, inplace=False)
            (1): Linear(in_features=1024, out_features=20, bias=True)
            (2): ReLU()
        )
        (locs): Sequential(
            (0): Dropout(p=0.0, inplace=False)
            (1): Linear(in_features=20, out_features=10, bias=True)
            (2): ReshapeLast()
        )
        (scales): Sequential(
            (0): Dropout(p=0.0, inplace=False)
            (1): Linear(in_features=20, out_features=10, bias=True)
            (2): Softplus(beta=1, threshold=20)
            (3): ReshapeLast()
        )
    )
)
(cv_model): VarianceReduction()
)

```

```

[46]: log = train_vae(
    model=model,
    opts=opts,
    training_data=train_ds,
    dev_data=val_ds,
    batch_size=256,
    num_epochs=5, # use more for better models
    check_every=100,
    sample_size_training=1,
    sample_size_eval=1,
    grad_clip=5.,
    device=my_device
)
0%|          | 0/1155 [00:00<?, ?it/s]

```

```
[47]: log.keys()
```

```
[47]: dict_keys(['dev.ELBO', 'dev.D', 'dev.R', 'dev.L', 'training.loss', 'training.ELBO',
              'training.D', 'training.R', 'training.cv_loss'])
```

```
[48]: fig, axs = plt.subplots(1, 3 + int('training.cv_loss' in log), sharex=True, sharey=False,
                           figsize=(20, 5))

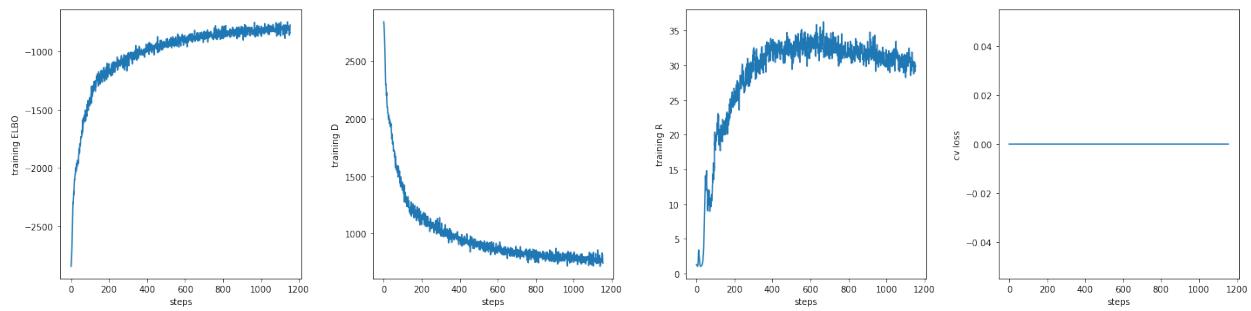
_ = axs[0].plot(np.array(log['training.ELBO'])[:,0], np.array(log['training.ELBO'])[:,1])
_ = axs[0].set_ylabel("training ELBO")
_ = axs[0].set_xlabel("steps")

_ = axs[1].plot(np.array(log['training.D'])[:,0], np.array(log['training.D'])[:,1])
_ = axs[1].set_ylabel("training D")
_ = axs[1].set_xlabel("steps")

_ = axs[2].plot(np.array(log['training.R'])[:,0], np.array(log['training.R'])[:,1])
_ = axs[2].set_ylabel("training R")
_ = axs[2].set_xlabel("steps")

if 'training.cv_loss' in log:
    _ = axs[3].plot(np.array(log['training.cv_loss'])[:,0], np.array(log['training.cv_loss'])[:,1])
    _ = axs[3].set_ylabel("cv loss")
    _ = axs[3].set_xlabel("steps")

fig.tight_layout(h_pad=2, w_pad=2)
```



```
[49]: fig, axs = plt.subplots(1, 4, sharex=True, sharey=False, figsize=(20, 5))

_ = axs[0].plot(np.array(log['dev.ELBO'])[:,0], np.array(log['dev.ELBO'])[:,1])
_ = axs[0].set_ylabel("dev ELBO")
_ = axs[0].set_xlabel("steps")

_ = axs[1].plot(np.array(log['dev.D'])[:,0], np.array(log['dev.D'])[:,1])
_ = axs[1].set_ylabel("dev D")
_ = axs[1].set_xlabel("steps")

_ = axs[2].plot(np.array(log['dev.R'])[:,0], np.array(log['dev.R'])[:,1])
_ = axs[2].set_ylabel("dev R")
_ = axs[2].set_xlabel("steps")

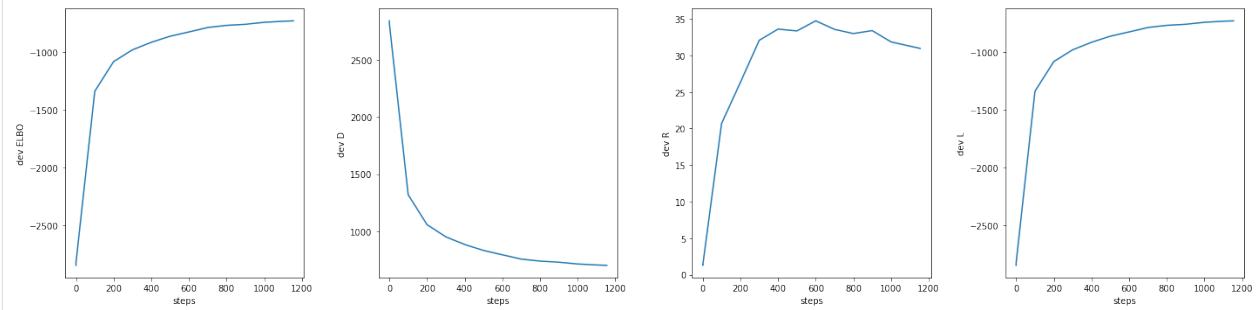
_ = axs[3].plot(np.array(log['dev.L'])[:,0], np.array(log['dev.L'])[:,1])
_ = axs[3].set_ylabel("dev L")
```

(continues on next page)

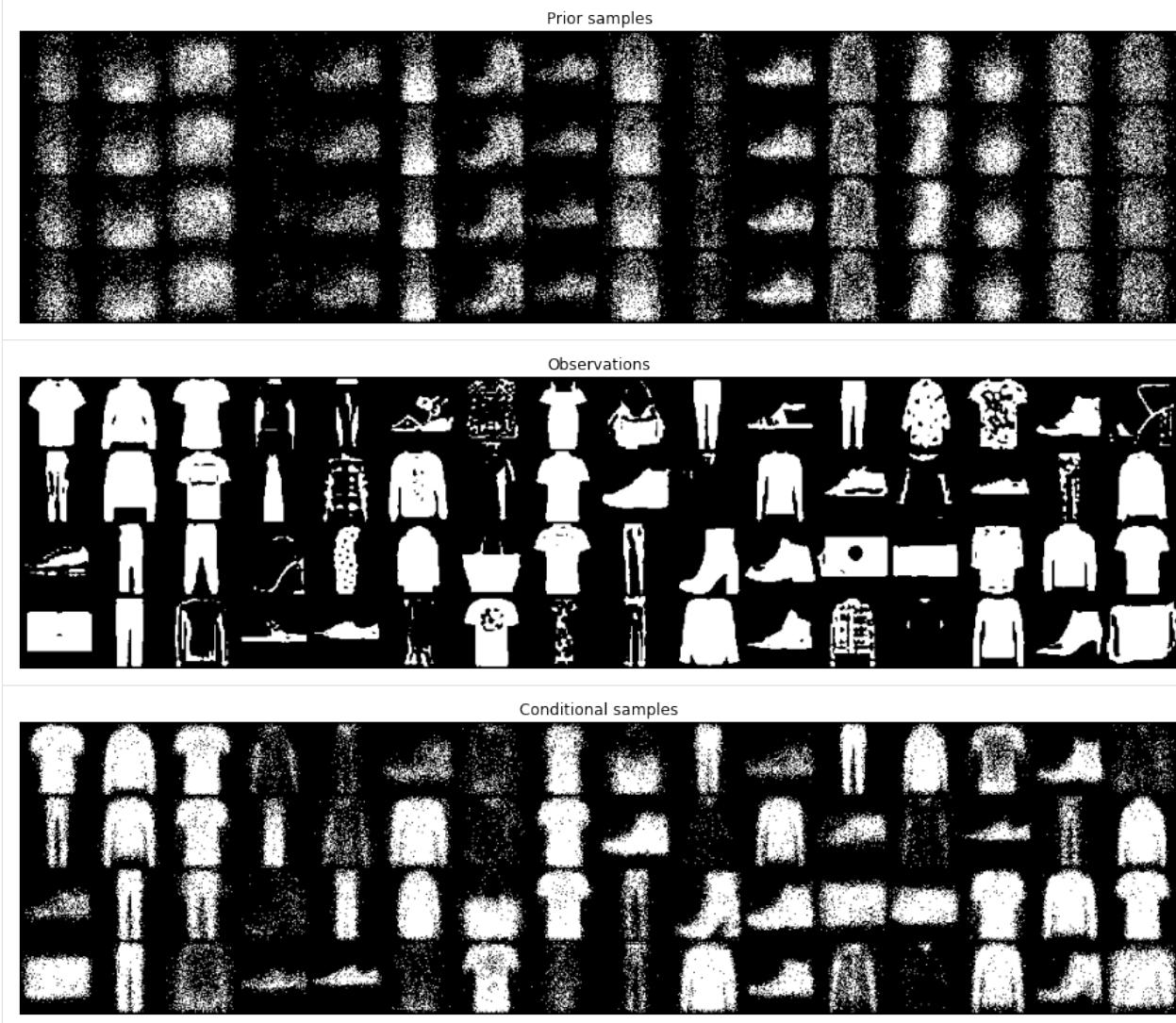
(continued from previous page)

```
_ = axs[3].set_xlabel("steps")
```

```
fig.tight_layout(h_pad=2, w_pad=2)
```



```
[50]: inspect_lvm(model, DataLoader(val_ds, 64, num_workers=2, pin_memory=True), my_device)
```



4.34.5 4. Beyond

There are various things you can try.

You can try to use a **trainable prior**. If you do, you will probably note that it is not trivial how to get the prior to be used in an interesting way. In fact, trained priors are completely data-driven, and there's no reason to believe that an NN will find a “data-driven” explanation of the data that is anything like what you would like it to. If you want the different components of your prior to specialise to certain types of output, you will need to design stronger pressures. For example, you may use some degree of annotation to inform what each component should typically be responsible for. Ideas that dispense with the need for annotation will have to focus on the architecture of the decoder or on other penalties in the loss. For example, a decoder that is built with some geometric properties.

If you use a trainable prior, it is a good idea to try and visualise what the final prior looks like. You can try sampling from it and plotting histograms of the different coordinates of the samples. You can flatten the samples and inspect the coordinates marginally, you can also use other plotting tools (see [some examples from seaborn](#)) to spot dependency, for example. And, of course, you can always use tools for dimensionality reduction (eg, t-SNE).

You can try to use a **stronger posterior approximation** that is reparameterisable. A good idea is to build a bijective transformation (ie., a normalising flow).

You can try to **improve the gradient estimator** of the mixture of mean field families. Within a mixture of C components that are each reparameterisable, only the component assignment is a discrete operation, so with access to the sampling procedure internal to mixture, one can design a customised gradient estimator that updates $\omega(x; \lambda)$ through SFE, but updates $\mu(x; \lambda)$ and $\sigma(x; \lambda)$ through reparameterisation.

4.35 AGM - Advanced Topics in Normalizing Flows - 1x1 convolution

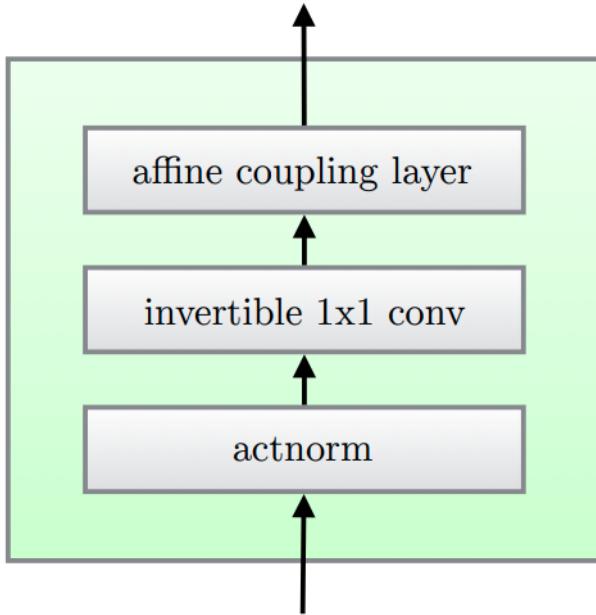
Filled notebook:

Authors: Cyril Hsu

4.35.1 Introduction

The [Glow](#), a flow-based generative model extends the previous invertible generative models, [NICE](#) and [RealNVP](#), and simplifies the architecture by replacing the reverse permutation operation on the channel ordering with **Invertible 1x1 Convolutions**. Glow is famous for being the one of the first flow-based models that works on high resolution images and enables manipulation in latent space. Let's have a look at the interactive [demonstration](#) from OpenAI.

Glow consists of a series of steps of flow. Each step of flow comprises **Actnorm** followed by an **Invertible 1x1 Convolution**, and finally a **Coupling Layer**.



Actnorm performs an affine transformation with a scale and bias parameter per channel, similar to that of batch normalization, but works on mini-batch size 1. The statistics (mean and std), however, are only calculated once to initialize the scale and bias parameters.

Invertible 1×1 Convolution with equal number of input and output channels is a generalization of any permutation of the channel ordering. Recall the operation between layers of the RealNVP flow, the ordering of channels is switched so that all the data dimensions have a chance to be mixed. 1x1 convolution is proposed to replace this fixed permutation with a learned invertible operation.

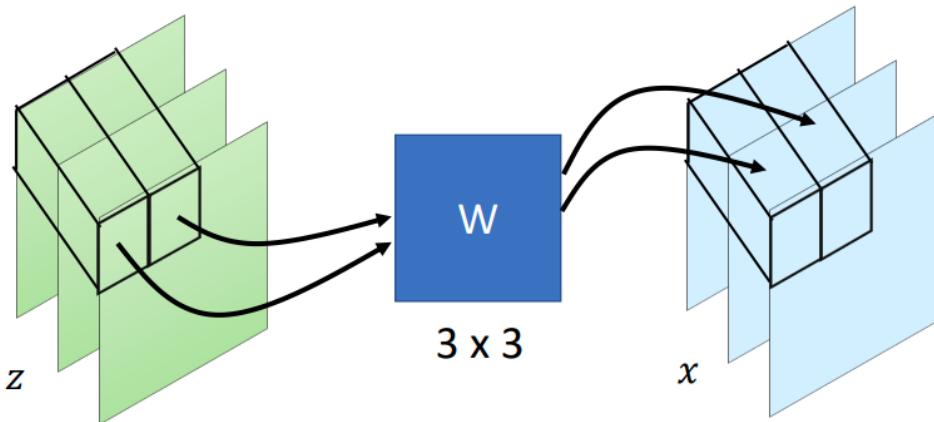
Coupling Layer is a powerful reversible transformation where the forward function, the reverse function and the log-determinant are computationally efficient. The design is the same as in RealNVP.

Description	Function	Reverse Function	Log-determinant
Actnorm.	$\forall i, j : \mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$	$\forall i, j : \mathbf{x}_{i,j} = (\mathbf{y}_{i,j} - \mathbf{b})/\mathbf{s}$	$h \cdot w \cdot \text{sum}(\log \mathbf{s})$
Invertible 1×1 convolution. $\mathbf{W} : [c \times c]$.	$\forall i, j : \mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}$	$\forall i, j : \mathbf{x}_{i,j} = \mathbf{W}^{-1}\mathbf{y}_{i,j}$	$h \cdot w \cdot \log \det(\mathbf{W}) $ or $h \cdot w \cdot \text{sum}(\log \mathbf{s})$
Affine coupling layer.	$\mathbf{x}_a, \mathbf{x}_b = \text{split}(\mathbf{x})$ $(\log \mathbf{s}, \mathbf{t}) = \text{NN}(\mathbf{x}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{y}_a = \mathbf{s} \odot \mathbf{x}_a + \mathbf{t}$ $\mathbf{y}_b = \mathbf{x}_b$ $\mathbf{y} = \text{concat}(\mathbf{y}_a, \mathbf{y}_b)$	$\mathbf{y}_a, \mathbf{y}_b = \text{split}(\mathbf{y})$ $(\log \mathbf{s}, \mathbf{t}) = \text{NN}(\mathbf{y}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{x}_a = (\mathbf{y}_a - \mathbf{t})/\mathbf{s}$ $\mathbf{x}_b = \mathbf{y}_b$ $\mathbf{x} = \text{concat}(\mathbf{x}_a, \mathbf{x}_b)$	$\text{sum}(\log(\mathbf{s}))$

In this tutorial, we will be focusing on the implementation of invertible 1x1 convolution layer.

4.35.2 Invertible 1x1 convolution

Given an input of shape $H \times W \times C$ applied with a 1×1 convolution with C filters, meaning the output tensor shape is also going to be $H \times W \times C$. Thus, each layer has a set of weights W with $C \times C$ values. The forward operation acts just like a typical convolution, while the inverse operation can be computed by simply applying a convolution with W^{-1} weights.



Enough descriptions! Now let's take a look at the code.

```
[14]: import torch
from torch import nn
from torch.nn import functional as F

class InvConv2d(nn.Module):
    def __init__(self, in_channel):
        super().__init__()

        weight = torch.randn(in_channel, in_channel)
        # use the Q matrix from QR decomposition as the initial weight to make sure it's
        ↪invertible
        q, _ = torch.qr(weight)
        weight = q.unsqueeze(2).unsqueeze(3)
        self.weight = nn.Parameter(weight)

    def forward(self, input, logdet, reverse=False):
        _, _, height, width = input.shape

        # You can also use torch.slogdet(self.weight)[1] to summarize the operations
        ↪below\n",
        dlogdet = (
            height * width * torch.log(torch.abs(torch.det(self.weight.squeeze())))
        )

        if not reverse:
            out = F.conv2d(input, self.weight)
            logdet = logdet + dlogdet

        else:
            out = F.conv2d(input, self.weight.squeeze().inverse().unsqueeze(2).
            ↪unsqueeze(3))
```

(continues on next page)

(continued from previous page)

```

logdet = logdet - dlogdet

return out, logdet

```

Note that to calculate the determinant of W could be computationally expensive, thus there's also an [implementation](#) which utilizes LU decomposition to speed up, as suggested in the Glow paper.

The idea is to parameterizing W directly in its LU decomposition:

$$W = PL(U + \text{diag}(s)),$$

where P is a permutation matrix, L is a lower triangular matrix with ones on the diagonal, U is an upper triangular matrix with zeros on the diagonal, and s is a vector.

The log-determinant is then simply:

$$\log |\det(W)| = \sum (\log |s|)$$

Please check out the link above for the implementation.

A small pitfall

As you might notice, there's an inverse operation for the weight W involved when the **1x1 convolution** is forwarding reversely. As a result, an error can occur when the weight W is not invertible, even though it seldom happens.

To our best knowledge, there's no elegant solution to address this, but an easy way to workaround: If this happens unfortunately during the training, one can try to restart from the recent checkpoint.

A complete flow block

Now we have the **Invertible 1x1 Convolution**. Together with the aforementioned **Actnorm** and **Coupling Layer**, we are ready to try out the power of the Glow by plugging the block into the model we had in the NFs tutorial!

```
[21]: class ActNorm(nn.Module):
    def __init__(self, in_channel):
        super().__init__()

        self.loc = nn.Parameter(torch.zeros(1, in_channel, 1, 1))
        self.log_scale = nn.Parameter(torch.zeros(1, in_channel, 1, 1))
        self.register_buffer("initialized", torch.tensor(0, dtype=torch.uint8))

    def initialize(self, input):
        with torch.no_grad():
            flatten = input.permute(1, 0, 2, 3).contiguous().view(input.shape[1], -1)
            mean = (
                flatten.mean(1)
                .unsqueeze(1)
                .unsqueeze(2)
                .unsqueeze(3)
                .permute(1, 0, 2, 3)
            )
            std = (
```

(continues on next page)

(continued from previous page)

```

        flatten.std(1)
        .unsqueeze(1)
        .unsqueeze(2)
        .unsqueeze(3)
        .permute(1, 0, 2, 3)
    )

    self.loc.data.copy_(-mean)
    self.log_scale.data.copy_(-std.clamp_(min=1e-6).log())

def forward(self, input, logdet, reverse=False):
    _, _, height, width = input.shape

    if self.initialized.item() == 0:
        self.initialize(input)
        self.initialized.fill_(1)

    dlogdet = height * width * torch.sum(self.log_scale)

    if not reverse:
        logdet += dlogdet
        return self.log_scale.exp() * (input + self.loc), logdet

    else:
        dlogdet *= -1
        logdet += dlogdet
        return input / self.log_scale.exp() - self.loc, logdet

```

A sample code for **Actnorm** is provided above.

As for **Coupling Layer**, please refer to the one in the NFs tutorial.

4.35.3 Conclusion

We've learned an advanced flow-based layer from the Glow model, an **Invertable 1x1 convolution**, which is adapted from the typical 1x1 convolution layers.

References

- Glow: Generative Flow with Invertible 1x1 Convolutions
- Glow: Better Reversible Generative Models
- <https://github.com/rosinality/glow-pytorch>
- Materials from NTU Speech Lab

4.36 HDL - Introduction to HyperParameter Tuning

Filled notebook:

Hyper-parameter tuning repository:

Authors: Samuele Papa

4.36.1 Introduction

This tutorial is meant as a description of the structure and features of the template GitHub repository to perform large scale hyperparameter tuning on a SLURM-based cluster using a combination of Pytorch Lightning, Hydra, Ax, MLFlow and Submitit.

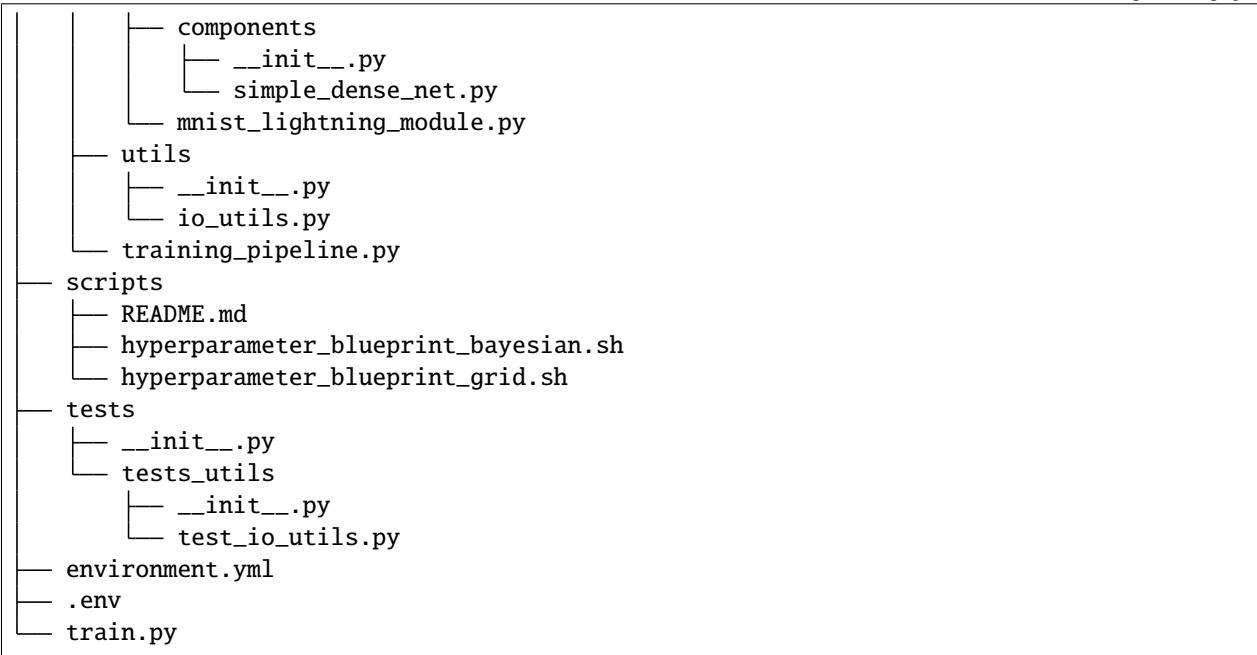
The template is not meant to be the definitive way hyperparameter tuning should be performed. Instead, it is meant to be a very good example from which to pick the elements and the structure that make most sense for your own future projects. For example, MLFlow is not very good at comparing images from multiple runs, and if a qualitative evaluation is necessary, then it would be a good idea to include Tensorboard as an additional logging library. The use of SLURM was dictated by the popularity of the system and the fact that it is in use on the surfsara cluster, but the same template would work for other systems. Many more considerations and adaptations could be made at the discretion of the researcher.

4.36.2 The structure

```
.  
├── hyperparameter_searcher  
│   ├── __init__.py  
│   └── config  
│       ├── data  
│       │   └── mnist_config.py  
│       ├── launcher  
│       │   └── launcher_config.py  
│       ├── logging  
│       │   └── logging_config.py  
│       ├── model  
│       │   └── mnist_module_config.py  
│       ├── sweeper  
│       │   └── sweeper_config.py  
│       ├── trainer  
│       │   └── trainer_config.py  
│       ├── train_bayesian_config.py  
│       └── train_grid_config.py  
└── data  
    ├── __init__.py  
    ├── dataloaders.py  
    └── mnist_datamodule.py  
└── loggers  
    ├── __init__.py  
    ├── loggers.py  
    └── mlflow_utils.py  
└── networks  
    └── __init__.py
```

(continues on next page)

(continued from previous page)



The main file of the whole repository is `train.py` in the root folder. From here the training procedure can run, and this is also the primary point that will be used for debugging the code. In the `tests` folder the eventual tests for the code being written will be placed, the `scripts` folder contains the scripts that we will be using for the hyperparameter search, and the `hyperparameter-searcher` folder is where the source code for all our experiments resides.

In this template, the source code for the experiments is divided into Python modules, that can be thought of as building components used to run the entire thing. Besides the modules, there is a file called `training_pipeline.py` which is where the training is defined, and that will be the only entry point to use the various modules from. In this template, only this file is present, however, often we will want to perform some additional analysis to our models after training, say creating nice visualizations with samples if we have a generative model, or computing a downstream task using the representations obtained. When this is the case, we will just need to add a new file, e.g. called `evaluation.py`, that will benefit from the already-defined modules and be used to load the model and run the evaluation.

The modules. In the experiment source code we have several modules. We can think of them as independent (as much as reasonable) packages that are used to perform a specific and complex task that can be re-used more than once or that is completely logically separate from other modules. This modular approach that separates based on function and not based on experiment, forces us to think of code that can be used immediately by all experiments we will be running, and is easier to maintain in the future.

Next, we will be discussing the different components that we are using for config management, logging and hyperparameter tuning.

4.36.3 Hydra

Hydra configurations are usually defined through `.yaml` files. However, we can also define them manually using Python. By using Python-based config files we have more freedom in the definition of the configurations and in the re-usability of the code. The trade-off is more complexity in the management of the code, as all the configuration needs to be defined manually.

Hydra interfaces with your scripts using a decorator to the main function. This defines where to get the configs from, and which primary config file should be used to parse the arguments. In the case of this repository, the configuration is done through Python, so we don't need the `config_path`:

```
@hydra.main(config_path=None, config_name=os.environ["MAIN_CONFIG"])
```

The primary config file is the entry point for your configuration. Here, you define the command line parameters that you support and which default configurations are to be used. A default configuration, as the name suggests, is a file that contains some default arguments. In our case, these will be handy for defining default configurations of the various datasets and models that we support.

From command line, all already-defined parameters can be changed, as well as new ones added. By default, you need to explicitly ask for an argument to be added, if this is not already defined in the config.

Overall, using Hydra is a very straightforward way to neatly organize your configurations and get closer to reproducible results.

Some observations

Hydra is used to manage the configuration of your experiments. All command line arguments and their processing can be handled through it. There are several advantages over using the traditional argument parser from Python. The first one is that we can more easily store and restore argument configurations. Another is that target classes can be defined directly in the configuration. A target class can then be initialized with the arguments given in the configuration. Think of what would happen if you wanted to switch between using `model_A` and `model_B` which are defined with the class `ModelA` and `ModelB`. From the config, you would say `model=model_A` and then in the code you would need a long if statement chain to select which class to launch with the given configuration, which in this case would be `ModelA`. Then, there would be several default parameters for this class that we would want to use, but would be hidden in the code. Instead, with Hydra you can more simply define the target class directly in the config file, which will be automatically selected when `model=model_A` is called in the command line. This will come automatically with all the parameters defined explicitly in the config file.

This only makes sense because the modular approach to config management allows for simple parameter switching when testing different models, datasets or when running different experiments entirely. Having modular configuration management simplifies the entire file structure as well, removing the need for separating different experiments in different folders, which can quickly become difficult to maintain as time goes on. Instead, each part of your project can be seen as a different package, one dedicated for data handling, one for model definition, one for the logging, another for visualization and maybe also one for all the metrics that you want to test your models with. Such modularity would be very difficult without also having modular configurations, which Hydra handles very easily.

One thing that is important to highlight is that Hydra is not some magical wand that we can use to solve all our problems. Instead, Hydra takes its root in the elegant configuration management that [Omegaconf](#) already provides. Hydra is a handy extension of Omegaconf, with some features tailored for machine learning. When more complicated things need to be done, do not hesitate to put your hands on what Hydra is doing and add your own code to make your workflow faster. Often, trying to work around the issue and use only the features available in the library slows you down more than you think.

4.36.4 MLFlow

MLFlow is a logging library, which is characterized by centralization of the logging, as to ease the process when multiple nodes are being used. Additionally, it provides a simplified way to compare the parameters that have been changed between runs, to get a quick overview of which change has made the most decisive impact in the performance of the model.

MLFlow interfaces with the code through PyTorch Lightning. When the `Trainer` is instantiated, among the loggers passed is MLFlow. In the code, this will seem a bit opaque, as the initialization of the loggers is done through Hydra's `instantiate` (in file `training_pipeline.py`):

```
mlflow_logger = hydra.utils.instantiate(logger)
loggers.append(mlflow_logger)
```

Another important aspect is the **checkpoint**, which allows to store useful information and the model's weights through MLFlow as well. Using MLFlow in the checkpointing process is useful to centralize all the information (in file `training_pipeline.py`):

```
callbacks.append(
    hydra.utils.instantiate(callback, mlflow_logger=mlflow_logger)
)
```

Some observations

MLFlow is an excellent logging tool to keep track of your experiments. Where MLFlow shines is in its ability to quickly compare multiple runs of an hyperparameter search. Also, it centralizes everything, putting all of the things you need in a single location, which is extremely handy when running large scale experiments.

There are a few downsides to MLFlow, as it is not excellent with image logging and is overall lacking in the ability to compare different metrics and the qualitative performance of different models. When this is necessary for your models (which is the often the case for computer vision tasks) MLFlow needs to be supported by additional loggers, such as Tensorboard. This is easily done with Pytorch Lightning, just asking for multiple loggers in the Trainer.

4.36.5 Hyperparameter Search

Here, we will be using **Hydra** (with the [Ax plugin](#)) and **Submitit** to generate the jobs and submit them to the SLURM cluster.

These are the most fundamental parameters that need to be configured based on how many resources you have available to perform the search, that will be defined from the `.sh` script (code from `launcher_config.py` file):

```
@dataclass
class SlurmConfig(SlurmQueueConf):
    partition: str = "gpu_titanrtx_shared"
    gpus_per_node: int = 1
    tasks_per_node: int = 1
    cpus_per_task: int = 6
    mem_gb: int = 60
    nodes: int = 1
    timeout_min: int = 1200 # how long can the job run
    array_parallelism: int = 2 # how many jobs can run simultaneously
```

The actual parameter search itself can be done in a *grid* or *bayesian* form. In the grid form, all combinations of the parameters are searched through, while in the bayesian form, the **Ax** sweeper will look for the best parameters to optimize a given metric, defined in the `optimized_metric` parameter. The syntax for defining what parameters should be *swept* can be found in this [Hydra documentation](#). The most fundamental ones are the following:

Choice. Select one between different options (here the example is different activation functions):

```
activation=choice(linear,relu)
```

Range. Sweeps the defined range based on the given step (here the example is a different lambda weight for regularization):

```
lambda=range(start=0,stop=10,step=2) # 0,2,4,6,8
```

Some observations

Hyperparameter tuning is a task that requires an incredible amounts of resources. Always consider the computing time and available resources before starting large computational studies. Try with smaller tests, trying to understand if the model is working and what is the rough range of parameters before continuing.

4.36.6 Conclusion

We have seen how we have used a combination of Hydra, MLFlow, SubmitIt, and the Ax plugin in Hydra to perform bayesian or grid hyperparameter searches in a SLURM-based cluster. We have seen how it interfaces with a simple project and have observed some of the strength and pitfalls of the methods.

It is crucial to remember that this setup is meant as a guide, to introduce the useful tools that you may want to use and how they interface together. Ultimately, the best fit for any case will be determined by the specific circumstances that you are facing.

References

Reference Repository. <https://github.com/NKI-AI/hyperparameter-search-template>

Hydra. <https://hydra.cc/>

MLFlow. <https://www.mlflow.org/docs/latest/index.html>

Pytorch Lightning. <https://www.pytorchlightning.ai/>

Ax Sweeper. https://hydra.cc/docs/next/plugins/ax_sweeper/

4.37 HDL - Introduction to Multi GPU Programming

Filled notebook:

Tutorial script files:

Recordings:

Authors: Samuele Papa

4.37.1 Introduction

Using multiple GPUs is a central part in scaling models to large datasets and obtain state of the art performance.

We have seen that, to control multiple GPUs, we need to understand the concepts of distributed computing. The core problem in distributed computing is the communication between nodes, which requires synchronization. Luckily, we are equipped with very limited communication tools, that minimize the chance that problems arise (the specifics are outside the scope of this course, to get more insight into the possible issues, look into [concurrent programming](#), race conditions, deadlocks, resource starvation, semaphores and barriers, and the book Operating Systems Internals and Design Principles).

Tutorial content. To better understand the primitives of communication in a distributed environment, we will begin by looking at some very basic exercises where simple operations are performed. Then, we will look at a more realistic

scenario, the computation of the loss in a one-layer classifier (*more* realistic, but still *very* simple). Finally, we will learn how to run full-scale training on multiple GPUs and multiple nodes using PyTorch Lightning.

Running the code

The code in these cells is not meant to be run with this notebook. Instead, the files provided should be used in an environment where multiple GPUs are available. This step is not required (all the outputs and explanation of the code are available here), but highly encouraged, as getting familiar with these concepts, especially the more simple primitives, will help when more cryptic errors start appearing in big projects.

Running the scripts can be done, for example, on the GPU partition of the LISA cluster ([General Knowledge on how to use the cluster](#)). After getting access using `ssh` (use WSL on Windows), we can setup the `conda` environment, by using the `module` package to load the correct anaconda version and then creating the environment based on the `environment.yml` file.

To upload the code, the `rsync` command can be used (on single files, it is possible to do it on folders by adding the `-r` option):

```
rsync file account@lisa-gpu.surfsara.nl:~/file
```

Then, the Anaconda module can be loaded and the environment created using:

```
module load 2021
module load Anaconda3/2021.05
conda env create -f environment.yml
```

It will take some time to download the necessary packages.

The main code to run is the following:

```
srun -p gpu_shared -n 1 --ntasks-per-node 1 --gpus 2 --cpus-per-task 2 -t 1:00:00 --pty
↪/bin/bash
```

where with `-p gpu_shared` we ask for the shared partition where there are GPUs available (other gpu partitions available are listed [here](#)), then, we specify that we will be running only 1 task in this node, we want 2 GPUs and we use 2 CPUs as well, for 1 hour. The run consists of executing the command `/bin/bash` which starts a bash console on the node that we have been assigned. This allows for input of the necessary commands.

Once inside, we can activate the correct anaconda environment and start running the scripts. We need to make sure that both GPUs are exposed to the script, with the following syntax:

```
python my_script.py
```

For these examples we will make use of the straightforward interface provided by `PyTorch`, a good summary is available at the documentation page, where all the details of the functions are shown.

Some useful utilities

The following code will help in the running of the experiments, with some plotting functions and setup of the distributed environment.

```
import torch
import torch.distributed as dist
import matplotlib.pyplot as plt
import numpy as np
from pathlib import Path
from typing import Optional

def rank_print(text: str):
    """
    Prints a statement with an indication of what node rank is sending it
    """
    rank = dist.get_rank()
    # Keep the print statement as a one-liner to guarantee that
    # one single process prints all the lines
    print(f"Rank: {rank}, {text}.")

def disk(
    matrix: torch.Tensor,
    center: tuple[int, int] = (1, 1),
    radius: int = 1,
    value: float = 1.0,
):
    """
    Places a disk with a certain radius and center in a matrix. The value given to the
    disk must be defined.
    Something like this:
    0 0 0 0 0
    0 0 1 0 0
    0 1 1 1 0
    0 0 1 0 0
    0 0 0 0 0
    Arguments:
    - matrix: the matrix where to place the shape.
    - center: a tuple indicating the center of the disk
    - radius: the radius of the disk in pixels
    - value: the value to write where the disk is placed
    """
    device = matrix.get_device()
    shape = matrix.shape

    # generate the grid for the support points
    # centered at the position indicated by position
    grid = [slice(-x0, dim - x0) for x0, dim in zip(center, shape)]
    x_coords, y_coords = np.mgrid[grid]
    mask = torch.tensor(
```

(continues on next page)

(continued from previous page)

```

        ((x_coords / radius) ** 2 + (y_coords / radius) ** 2 <= 1), device=device
    )
matrix = matrix * (~mask) + mask * value

return matrix, mask

def square(
    matrix: torch.tensor,
    topleft: tuple[int, int] = (0, 0),
    length: int = 1,
    value: float = 1.0,
):
    """
    Places a square starting from the given top-left position and having given side_
    length.
    The value given to the disk must be defined.
    Something like this:
    0 0 0 0 0
    0 1 1 1 0
    0 1 1 1 0
    0 1 1 1 0
    0 0 0 0 0
    Arguments:
    - matrix: the matrix where to place the shape.
    - topleft: a tuple indicating the top-left-most vertex of the square
    - length: the side length of the square
    - value: the value to write where the square is placed
    """
    device = matrix.get_device()
    shape = matrix.shape
    grid = [slice(-x0, dim - x0) for x0, dim in zip(topleft, shape)]
    x_coords, y_coords = np.mgrid[grid]
    mask = torch.tensor(
        (
            (x_coords <= length)
            & (x_coords >= 0)
            & (y_coords >= 0)
            & (y_coords <= length)
        ),
        device=device,
    )
    matrix = matrix * (~mask) + mask * value

    return matrix, mask

def plot_matrix(
    matrix: torch.Tensor,
    rank: int,
    title: str = "Matrix",

```

(continues on next page)

(continued from previous page)

```

name: str = "image",
folder: Optional[str] = None,
storefig: bool = True,
):
    """
    Helper function to plot the images more easily. Can store them or visualize them
    right away.
    """

    plt.figure()
    plt.title(title)
    plt.imshow(matrix.cpu(), cmap="tab20", vmin=0, vmax=19)
    plt.axis("off")
    if folder:
        folder = Path(folder)
        folder.mkdir(exist_ok=True, parents=True)
    else:
        folder = Path(".")

    if storefig:
        plt.savefig(folder / Path(f"rank_{rank}_{name}.png"))
    else:
        plt.show()
    plt.close()

```

When starting the distributed environment, we need to decide a backend between `gloo`, `nccl` and `mpi`. The support for these libraries needs to be already available. The `nccl` backend should be already available from a GPU installation of PyTorch (CUDA Toolkit is required). On a Windows environment, only `gloo` works, but we will be running these scripts on a Unix environment.

The second fundamental aspect is how the information is shared between nodes. The method we choose is through a shared file, that is accessible from all the GPUs. It is important to remember that access to this file should be quick for all nodes, so on LISA we will put it in the `scratch` folder.

The other two parameters are the `rank` and `world_size`. The rank refers to the identifier for the current device, while the world size is the number of devices available for computation.

When setting up the distributed environment, the correct GPU device should be selected. For simplicity, we select the GPU that has ID corresponding to the rank, but this is not necessary.

Computation nodes could reside in different nodes, when this happens, using a shared fil

```

def setup_distrib(
    rank: int,
    world_size: int,
    init_method: str = "file:///scratch/sharedfile",
    backend: str = "nccl",
):
    # select the correct device for this process
    torch.cuda.set_device(rank)

    # initialize the processing group
    torch.distributed.init_process_group(
        backend=backend, world_size=world_size, init_method=init_method, rank=rank
    )

```

(continues on next page)

(continued from previous page)

```
# return the current device
return torch.device(f"cuda:{rank}")
```

Training a model on multiple GPUs is a clear example that we will keep in mind throughout this tutorial to contextualize how to make use of the available primitives of communication in distributed computing.

Initially, we want to give the current weights of the model to every GPU that we are using. To do so, we will **broadcast** the necessary tensors.

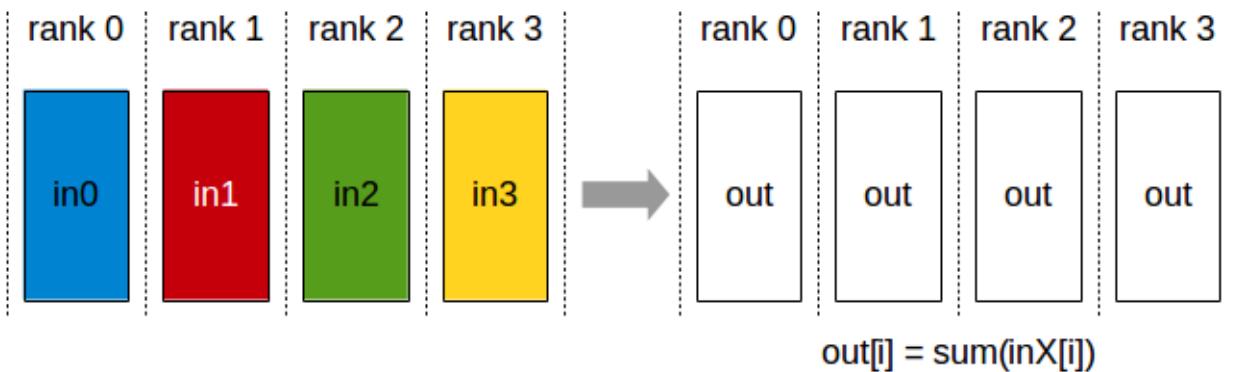
Then, each GPU will collect a subset of the full batch, lets say only 64 out of 256 samples, from memory and perform a forward pass of the model. At the end, we need to compute the loss over the entire batch of 256 samples, but no GPU can fit all of these. Here, the **reduction** primitive comes to the rescue. The tensors that reside in different GPUs are collected and an operation is performed that will *reduce* the tensors to a single one. This allows for the result of the operation to still fit in memory. We may want to keep this result in a single GPU (using **reduce**) or send it to all of them (using **all_reduce**).

The operations that we can perform are determined by the backend that we are currently using. When using `ncc1`, the list of available operations is the following: - SUM - AVG (only version 2.10 or higher) - PRODUCT - MIN - MAX

Communication Primitives

4.37.2 All Reduce

As we can see from the illustration, the **all reduce** primitive performs an operation between the tensors present in each GPU and replaces them with the result of the operation. The file to run is `all_reduce.py`.



```
from utils import setup_distrib, disk, square, rank_print, plot_matrix
import torch.distributed as dist
import torch.multiprocessing as mp
import torch

# operation performed by the reduce
OPERATION = dist.ReduceOp.MAX

def main_process(rank:int, world_size:int=2):
    device = setup_distrib(rank, world_size)
    rank_print("test")
    image = torch.zeros((11,11), device=device)
```

(continues on next page)

(continued from previous page)

```
if rank == 0:
    rank_image, rank_mask = disk(image, (3,3), 2, rank+1)
elif rank == 1:
    rank_image, rank_mask = square(image, (3,3), 2, rank+1)

plot_matrix(rank_image, rank, title=f"Rank {rank} Before All Reduce", name="before_all_reduce", folder="all_reduce")

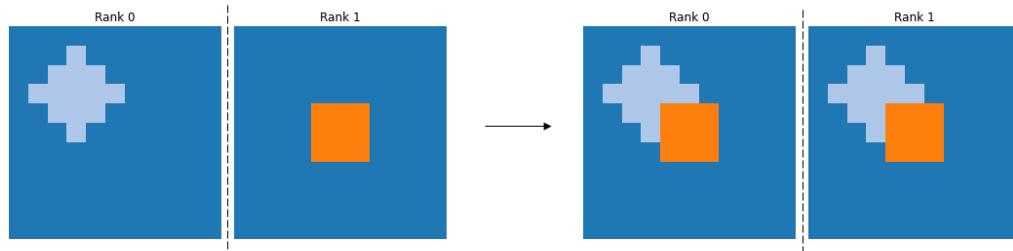
# The main operation
dist.all_reduce(rank_image, op=OPERATION)
plot_matrix(rank_image, rank, title=f"Rank {rank} After All Reduce Operation: {OPERATION}", name="after_all_reduce", folder="all_reduce")

if __name__ == '__main__':
    mp.spawn(main_process, nprocs=2, args=())
```

The result of this operation is illustrated in the figure below. The operation is performed between the tensors stored in the different devices and the result is spread across all devices.

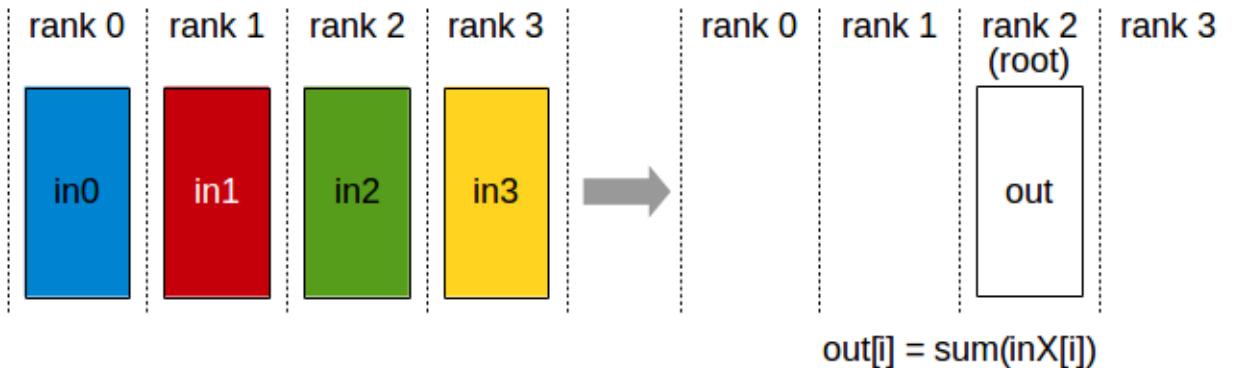
All Reduce

Operation: max



4.37.3 Reduce

As we can see from the illustration, the **reduce** primitive performs an operation between the tensors present in each GPU and sends the result only to the root rank. In Pytorch, we can define the destination rank. The file to run is `reduce.py`.



```
from utils import setup_distrib, disk, square, rank_print, plot_matrix
import torch.distributed as dist
import torch.multiprocessing as mp
import torch
DESTINATION_RANK = 0
OPERATION = dist.ReduceOp.MAX

def main_process(rank, world_size=2):
    device = setup_distrib(rank, world_size)

    image = torch.zeros((11,11), device=device)

    if rank == 0:
        rank_image, rank_mask = disk(image, (4,5), 2, rank+1)
    elif rank == 1:
        rank_image, rank_mask = square(image, (3,3), 2, rank+1)

    plot_matrix(rank_image, rank, title=f"Rank {rank} Before Reduce", name="before_reduce",
               folder="reduce")

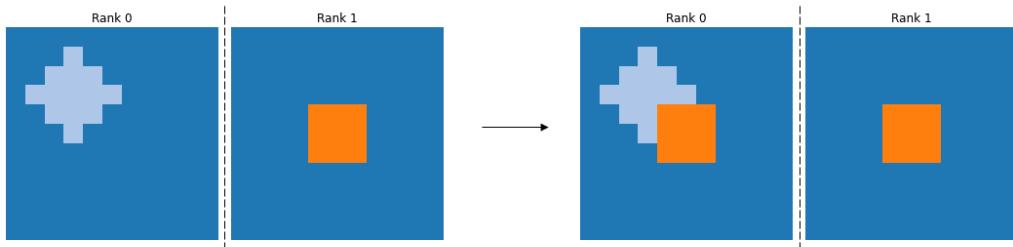
    # The main operation
    dist.reduce(rank_image, dst=DESTINATION_RANK, op=OPERATION)
    plot_matrix(rank_image, rank, title=f"Rank {rank} After Reduce Operation: {OPERATION}",
               name="after_reduce", folder="reduce")

if __name__ == '__main__':
    mp.spawn(main_process, nprocs=2, args=())
```

The results are shown below. We can see how only the rank 0, the one we selected, has the result of the operation. This helps in reducing the processing time, if the operation is executed in an asynchronous way, all other GPUs can keep processing while the root one is receiving the result.

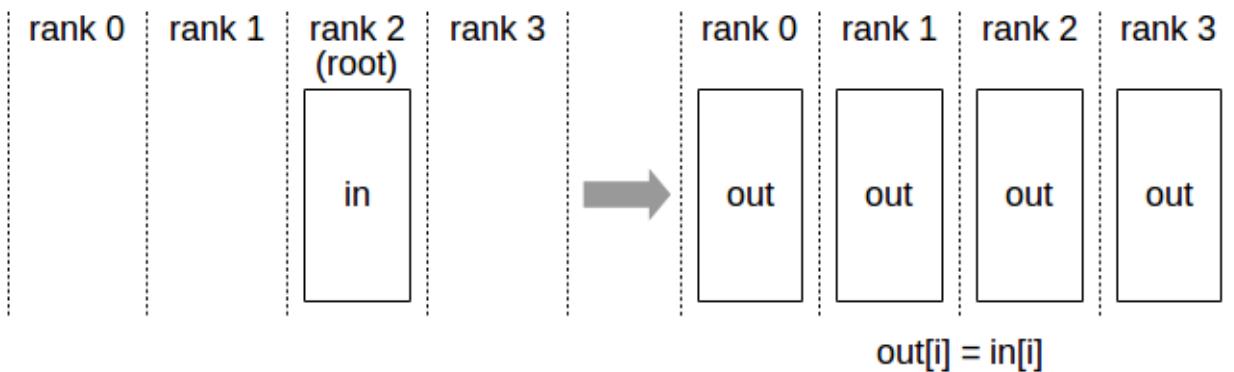
Reduce

Operation: max



4.37.4 Broadcast

The **broadcast** operation is fundamental, as it allows to send (broadcast) data from one GPU to all others in the network. The file to run is `broadcast.py`.



```
from utils import setup_distrib, disk, square, rank_print, plot_matrix
import torch.distributed as dist
import torch.multiprocessing as mp
import torch
SOURCE_RANK = 0
OPERATION = dist.ReduceOp.MAX

def main_process(rank, world_size=2):
    device = setup_distrib(rank, world_size)

    image = torch.zeros((11, 11), device=device)
```

(continues on next page)

(continued from previous page)

```

if rank == 0:
    rank_image, rank_mask = disk(image, (4,5), 2, rank+1)
elif rank == 1:
    rank_image, rank_mask = square(image, (3,3), 2, rank+1)

plot_matrix(rank_image, rank, name="before_broadcast", folder="broadcast")

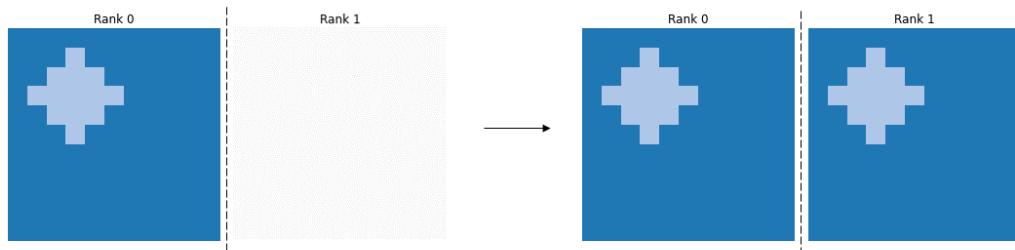
# The main operation
dist.broadcast(rank_image, src=SOURCE_RANK)
plot_matrix(rank_image, rank, name="after_broadcast", folder="broadcast")

if __name__ == '__main__':
    mp.spawn(main_process, nprocs=2, args=())

```

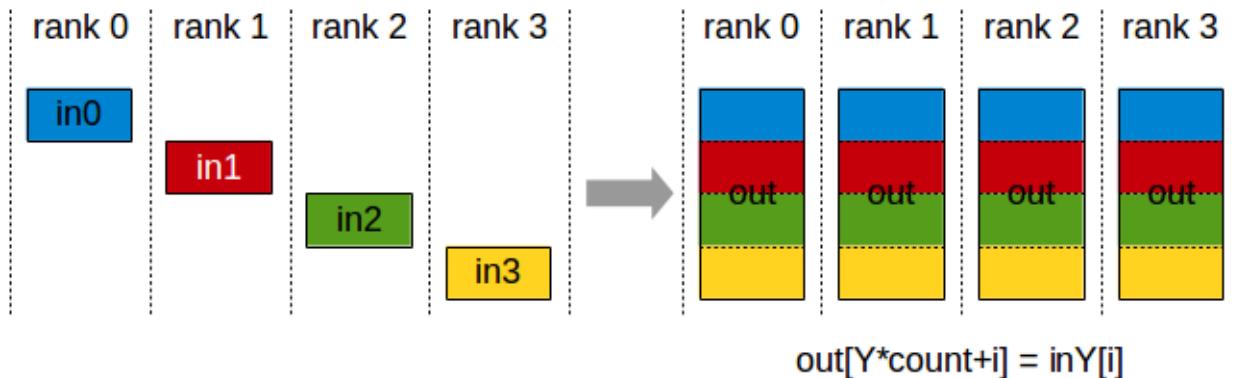
In this illustration we see how, the rank 1 GPU gets the correct image after the broadcast is performed.

Broadcast



4.37.5 All Gather

The **all gather** operation allows for all GPUs to have access to all the data processed by the others. This can be especially useful when different operations need to be performed by each GPU, after a common operation has been performed on each subset of the data. It is important to note that the entirety of the data needs to fit in a single GPU, so here the bottleneck won't be the memory, instead, it will be the processing speed. The file to run is `all_gather.py`.



```
from utils import setup_distrib, disk, square, rank_print, plot_matrix
import torch.distributed as dist
import torch.multiprocessing as mp
import torch
def main_process(rank, world_size=2):
    device = setup_distrib(rank, world_size)

    image = torch.zeros((11, 11), device=device)

    rank_images = []

    if rank == 0:
        rank_images.append(disk(image, (4, 5), 2, rank+1)[0])
    elif rank == 1:
        rank_images.append(disk(image, (7, 6), 2, rank+1)[0])

    output_tensors = []
    for _ in range(world_size):
        output_tensors.append(torch.zeros_like(image, device=device))

    plot_matrix(output_tensors[0], rank, title=f"Rank {rank}", name="before_gather_0",
    ↪ folder="all_gather")
    plot_matrix(output_tensors[1], rank, title=f"", name="before_gather_1", folder="all_"
    ↪ gather)

    # The main operation
    dist.all_gather(output_tensors, rank_images[0])
    plot_matrix(output_tensors[0], rank, title=f"Rank {rank}", name="after_gather_0",
    ↪ folder="all_gather")
    plot_matrix(output_tensors[1], rank, title=f"", name="after_gather_1", folder="all_"
    ↪ gather)
```

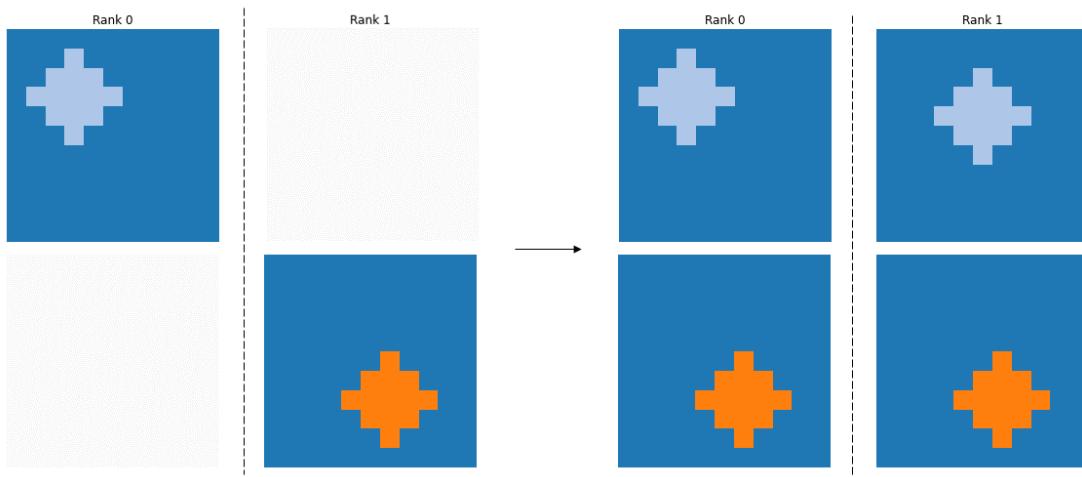
(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    mp.spawn(main_process, nprocs=2, args=())
```

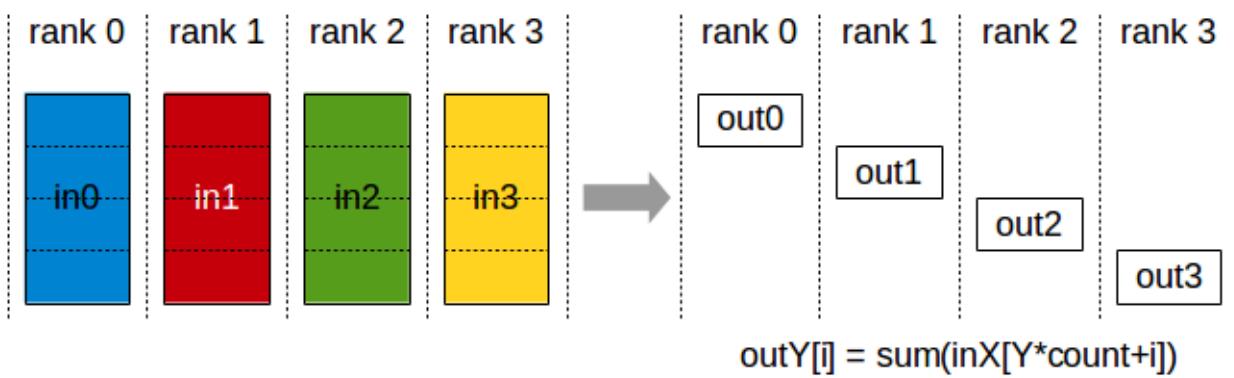
Here we can see the result of the **all_gather**. All GPUs now have access to the data that was initially only present in some of them.

All Gather



4.37.6 Reduce Scatter

With **reduce scatter** we can perform an operation on just a subset of the whole data and have each GPU have the partial results. The file to run is `reduce_scatter.py`.



```
from utils import setup_distrib, disk, square, rank_print, plot_matrix
import torch.distributed as dist
import torch.multiprocessing as mp
import torch
OPERATION = dist.ReduceOp.MAX
```

(continues on next page)

(continued from previous page)

```
def main_process(rank, world_size=2):
    device = setup_distrib(rank, world_size)

    image = torch.zeros((11,11), device=device)

    input_tensors = []

    if rank == 0:
        input_tensors.append(disk(image, (4,5), 2, rank+1)[0])
        input_tensors.append(square(image, (5,5), 3, rank+1)[0])
    elif rank == 1:
        input_tensors.append(disk(image, (7,6), 2, rank+1)[0])
        input_tensors.append(square(image, (0,2), 4, rank+1)[0])

    output = torch.zeros_like(image, device=device)

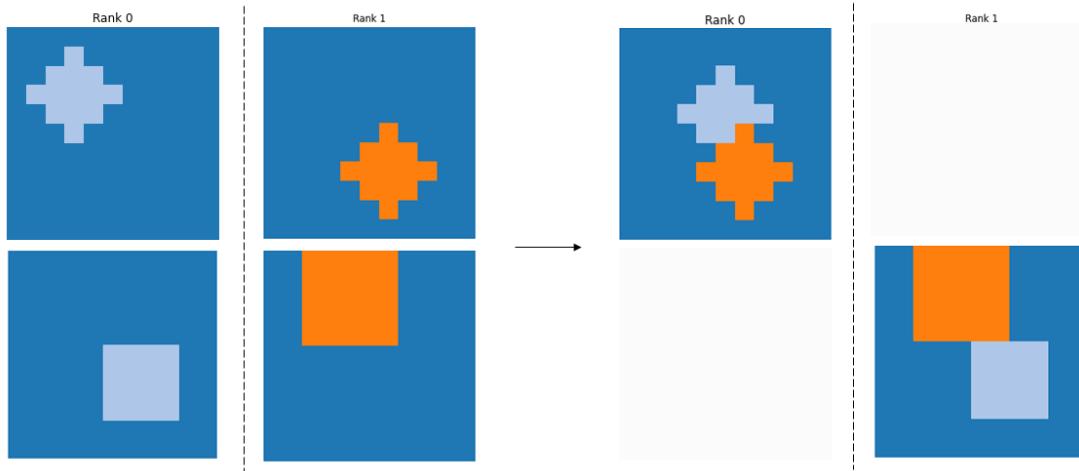
    plot_matrix(input_tensors[0], rank, title=f"Rank {rank}", name="before_reduce_scatter_0", folder="reduce_scatter")
    plot_matrix(input_tensors[1], rank, title=f"", name="before_reduce_scatter_1", folder="reduce_scatter")
    plot_matrix(output, rank, title=f"", name="before_reduce_scatter", folder="reduce_scatter")

    # The main operation
    dist.reduce_scatter(output, input_tensors, op=OPERATION)
    plot_matrix(output, rank, title=f"", name="after_reduce_scatter", folder="reduce_scatter")

if __name__ == '__main__':
    mp.spawn(main_process, nprocs=2, args=())
```

In this figure we see that only one image is available at the end, allowing for the operation to be performed across GPUs while keeping the overall final memory footprint low.

Reduce Scatter

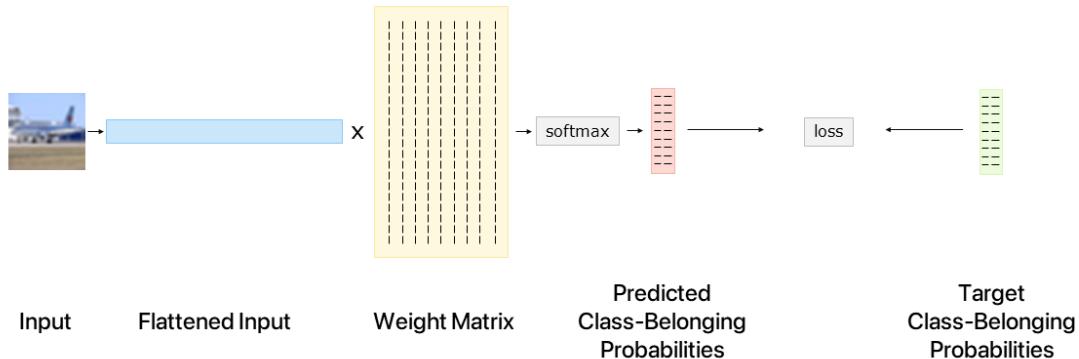


Exercise

An interesting thing to test, if you have access to a multi-GPU environment, is what are the physical limits of the system, and if the processing speed is the same with any number of tensors being loaded into the GPU. Is it more efficient to use a multiple of the number of cores that are processing the data in the GPU, or is the difference in performance negligible? You can investigate these topics through experimentation.

A more comprehensive example

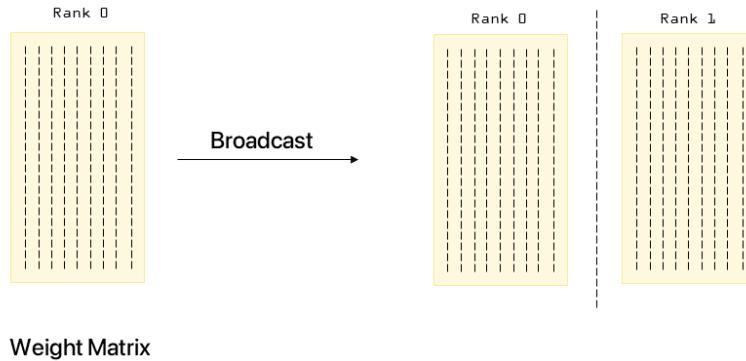
We will now look at a more realistic scenario (code in `single_layer.py`), the overall process is shown in the figure below.



The first thing we do is to spawn the two processes. In each, we begin by initializing the distributed processing environment.

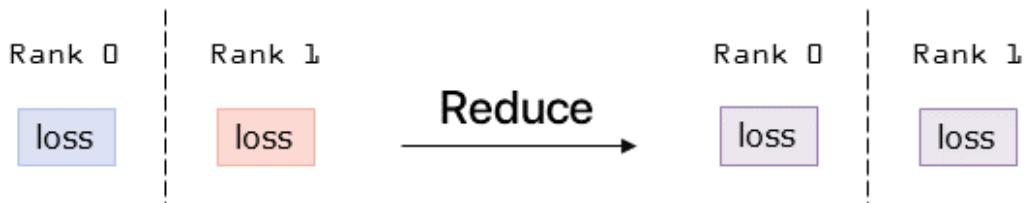
Then, the datasets needs to be downloaded. Here, I assume that it has not been downloaded yet, and I only let the GPU in rank 0 perform this operation. This avoids having two processes writing in the same file. In order to have the other process wait for the first one to download, a **barrier** is used. The working principle is very simple, when a barrier is reached in the code, the process waits for all other processes to also reach that point in the code. Here we see how this can be a very useful construct in parallel computing, all processes require the dataset to be downloaded before proceeding, so one of them starts the download, and all wait until it's done.

Then we initialize the weights, only in the rank 0 GPU, and **broadcast** them to all other GPUs. This broadcast operation is performed asynchronously, to allow for the rank 0 GPU to start loading images before the rank 1 has received the weights. This operation is akin to what DataParallel does, which is slowing the processing of the other GPUs down, waiting to receive the weights from the root GPU.



Each GPU will then load the images from disk, perform a product to find the activations of the next layer and calculate a softmax to get class-belonging probabilities.

Finally, the loss is computed by summing over the dimensions and a **reduction** with sum is performed to compute the overall loss over the entire batch.



```

import torch
import torch.distributed as dist
import torch.multiprocessing as mp
import torchvision
from torchvision import transforms
from utils import rank_print

DSET_FOLDER = "/scratch/"

def main_process(rank, world_size=2):
    print(f"Process for rank: {rank} has been spawned")

```

(continues on next page)

(continued from previous page)

```

# Setup the distributed processing
device = setup_distrib(rank, world_size)

# Load the dataset in all processes download only in the first one
if rank == 0:
    dset = torchvision.datasets.CIFAR10(DSET_FOLDER, download=True)

# Make sure download has finished
dist.barrier()

# Load the dataset
dset = torchvision.datasets.CIFAR10(DSET_FOLDER)

input_size = 3 * 32 * 32 # [channel size, height, width]
per_gpu_batch_size = 128
num_classes = 10
if dist.get_rank() == 0:
    weights = torch.rand((input_size, num_classes), device=device)
else:
    weights = torch.zeros((input_size, num_classes), device=device)

# Distribute weights to all GPUs
handle = dist.broadcast(tensor=weights, src=0, async_op=True)
rank_print(f"Weights received.")

# Flattened images
cur_input = torch.zeros((per_gpu_batch_size, input_size), device=device)

# One-Hot encoded target
cur_target = torch.zeros((per_gpu_batch_size, num_classes), device=device)
for i in range(per_gpu_batch_size):
    rank_print(f"Loading image {i+world_size*rank} into GPU...")
    image, target = dset[i + world_size * rank]
    cur_input[i] = transforms.ToTensor()(image).flatten()
    cur_target[i, target] = 1.0

# Compute the linear part of the layer
output = torch.matmul(cur_input, weights)
rank_print(f"\nComputed output: {output}, Size: {output.size()}.")

# Define the activation function of the output layer
logsoftmax = torch.nn.LogSoftmax(dim=1)

# Apply activation function to output layer
output = logsoftmax(output)
rank_print(f"\nLog-Softmaxed output: {output}, Size: {output.size()}.")

loss = output.sum(dim=1).mean()
rank_print(f"\nLoss: {loss}, Size: {loss.size()}")

# Here the GPUs need to be synched again

```

(continues on next page)

(continued from previous page)

```

dist.reduce(tensor=loss, dst=0, op=dist.ReduceOp.SUM)

rank_print(f"Final Loss: {loss/world_size}")

if __name__ == "__main__":
    mp.spawn(main_process, nprocs=2, args=())

```

PyTorch Lightning

When you have your Pytorch Lightning module defined, scaling to multiple GPUs and multi nodes is very straightforward (more details [here](#)):

```
trainer = Trainer(gpus=8, strategy="ddp")
```

Yes, seems impossible, but it's real. In most cases this is all you need to run multi GPU training.

Conclusion

We have seen how the basic collective primitives for communication work in a multi GPU environment. The reduction and broadcast operations are maybe the most important ones, allowing for delivery of data to all nodes and to perform mathematical operations on the data present in all the nodes.

We have seen how we can use these collectives to perform a calculation of the loss of a neural network, but the same can be extended to any type of parallelizable computation.

Finally, we saw how simple it is to set a PyTorch Lightning training to use multiple GPUs.

4.37.7 References

Pytorch Documentation on Distributed Communication. <https://pytorch.org/docs/stable/distributed.html>

NVIDIA NCCL Developer Guide. <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html>

PyTorch Lightning Multi-GPU Training. https://pytorch-lightning.readthedocs.io/en/stable/advanced/multi_gpu.html

Concurrent Programming and Operating Systems. Stallings, William. Operating Systems : Internals and Design Principles. Upper Saddle River, N.J. :Prentice Hall, 2001.

4.38 BNN - Comparing to Non-Bayesian Methods for Uncertainty

Filled notebook:

Empty notebook:

Authors: Ilze Amanda Auzina, Leonard Bereska and Eric Nalisnick

In this tutorial we will investigate what are some benefits of Bayesian Neural Networks (BNN) over point estimate Neural Networks.

Import the usual suspects...

```
[ ]: import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from tqdm.auto import trange, tqdm

torch.manual_seed(81);
```

4.38.1 Simulate Data

Let's simulate a wiggly line and draw observations in separated regions...

```
[ ]: # set Numpy seed
np.random.seed(42)

# sample observations
x = np.linspace(-.2, .2, 500)
x = np.hstack([x, np.linspace(.6, 1, 500)])
= 0.02 * np.random.randn(x.shape[0])
y = x + 0.3 * np.sin(2 * np.pi * (x + )) + 0.3 * np.sin(4 * np.pi * (x + )) +

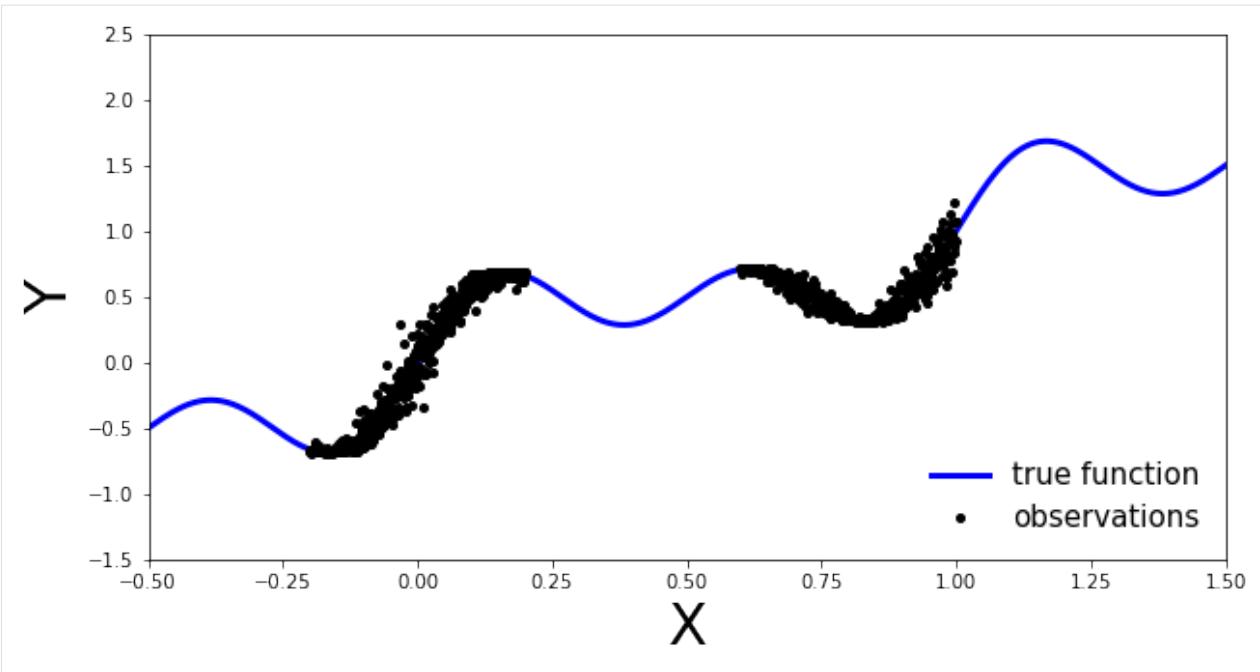
# compute true function
x_true = np.linspace(-.5, 1.5, 1000)
y_true = x_true + 0.3 * np.sin(2 * np.pi * x_true) + 0.3 * np.sin(4 * np.pi * x_true)

# make plot
xlims = [-.5, 1.5]
ylims = [-1.5, 2.5]

fig, ax = plt.subplots(figsize=(10, 5))
plt.xlim(xlims)
plt.ylim(ylims)
plt.xlabel("X", fontsize=30)
plt.ylabel("Y", fontsize=30)

ax.plot(x_true, y_true, 'b-', linewidth=3, label="true function")
ax.plot(x, y, 'ko', markersize=4, label="observations")

plt.legend(loc=4, fontsize=15, frameon=False);
```



As you can see, we have the true function in blue. The observations are observable in two regions of the function and there is some noise in their measurement. We will use this simple data to showcase the differences between BNN and deterministic NN.

4.38.2 Define non-Bayesian Neural Network

First let's create our point estimate neural network, in other words a standard fully connected MLP. We will define the number of hidden layers dynamically so we can reuse the same class for different depths. We will also add two flags, *residual* and *dropout* flag, this will allow us to easily use the same architecture for our BNN.

```
[ ]: class MLP(nn.Module):
    def __init__(self, in_dim=1, out_dim=1, hid_dim=10, n_hid_layers=1, residual_
    ↵flag=False, dropout_flag=False, dropout_prob = 0.5):
        super().__init__()
        assert in_dim > 0
        assert out_dim > 0
        assert hid_dim > 0
        assert n_hid_layers > 0
        self.activation = nn.Tanh() # could also be nn.ReLU()
        self.residual_flag = residual_flag
        self.dropout_flag = dropout_flag

        # optional: activate dropout
        self.dropout = nn.Dropout(p=dropout_prob)

        # dynamically define architecture
        self.layer_sizes = [in_dim] + n_hid_layers * [hid_dim] + [out_dim]
        layer_list = [nn.Linear(self.layer_sizes[idx-1], self.layer_sizes[idx]) for idx in_
        ↵range(1, len(self.layer_sizes))]
        self.layers = nn.ModuleList(layer_list)
```

(continues on next page)

(continued from previous page)

```

def forward(self, x, y=None):
    #x = x.reshape(-1, 1)
    # input --> hid
    x = self.activation(self.layers[0](x))

    for layer in self.layers[1:-1]:
        # hid --> hid
        x_temp = self.activation(layer(x))

        # optional: compute dropout mask
        if self.dropout_flag:
            x_temp = self.dropout(x_temp)

        # optional: compute residual connection
        if self.residual_flag:
            x = x_temp + x
        else:
            x = x_temp

    # hid --> output mean
    mu = self.layers[-1](x).squeeze()
    return mu

```

4.38.3 Train one deterministic NN

Training

Now let's train our MLP with the training data we generated above:

```

[ ]: # define model and data
mlp = MLP(hid_dim=30, n_hid_layers=2, residual_flag=True)
x_train = torch.from_numpy(x).float()[:, None]
y_train = torch.from_numpy(y).float()

[ ]: #select an optimizer and loss function
mlp_optimizer = torch.optim.Adam(params=mlp.parameters(), lr=1e-3)
mlp_criterion = nn.MSELoss()

# training loop
bar = trange(3000)
for epoch in bar:
    mlp_optimizer.zero_grad()
    loss = mlp_criterion(y_train, mlp(x_train))
    bar.set_postfix(loss=f' {loss / x_train.shape[0]:.3f}')
    loss.backward()
    mlp_optimizer.step()

0% | 0/3000 [00:00<?, ?it/s]

```

Evaluate

Let's investigate how our deterministic MLP generalizes over the entire domain of our input variable x (the model was only trained on the observations, now we will also pass in data outside this region)

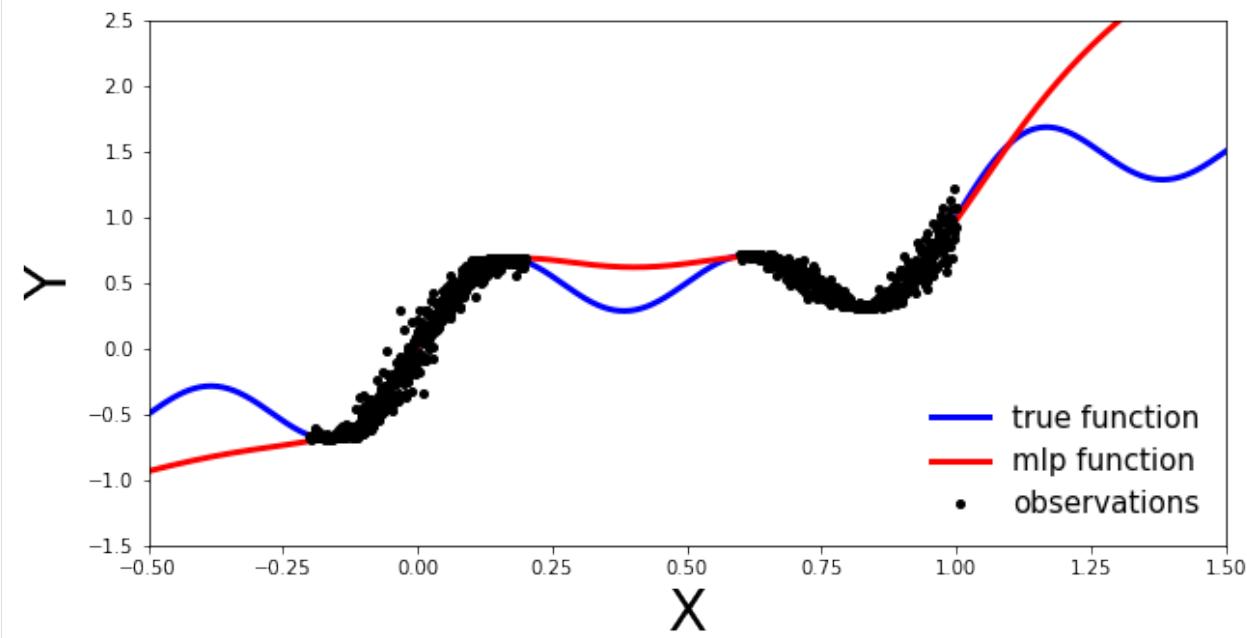
```
[ ]: # compute predictions everywhere
x_test = torch.linspace(xlims[0], xlims[1], 3000)[:, None]
y_preds = mlp(x_test).clone().detach().numpy();
```

```
[ ]: # generate plot
```

```
fig, ax = plt.subplots(figsize=(10, 5))
plt.xlim(xlims)
plt.ylim(ylims)
plt.xlabel("X", fontsize=30)
plt.ylabel("Y", fontsize=30)

ax.plot(x_true, y_true, 'b-', linewidth=3, label="true function")
ax.plot(x_test, y_preds, 'r-', linewidth=3, label="mlp function")
ax.plot(x, y, 'ko', markersize=4, label="observations")

plt.legend(loc=4, fontsize=15, frameon=False);
```



We can see that our deterministic MLP (red line) has correctly learned the data distribution in the training regions, however, as the model has not learned the underlying sinusoidal wave function, its predictions outside the training region are inaccurate. As our MLP is a point estimate NN we have no measure confidence in the predictions outside the training region. In the upcoming sections let's see how this compares to BNN.

4.38.4 Deep Ensemble

Deep ensembles were first introduced by Lakshminarayanan et al. (2017). As the name implies multiple point estimate NN are trained, *an ensemble*, and the final prediction is computed as an average across the models. From a Bayesian perspective the different point estimates correspond to modes of a Bayesian posterior. This can be interpreted as approximating the posterior with a distribution parametrized as multiple Dirac deltas:

$$q_{\phi}(\theta|D) = \sum_{\theta_i} \alpha_{\theta_i} \delta_{\theta_i}(\theta)$$

where : *math* : ‘ α_{θ_i} ’ are positive constants such that their sum is equal to one.

Training

We will reuse the MLP architecture introduced before, simply now we will train an ensemble of such models

```
[ ]: #define the size of the ensemble: number of NN to train
ensemble_size = 5
ensemble = []
mlp_optimizers = []

# initialize the ensemble
for _ in range(ensemble_size):
    net = MLP(hid_dim=30, n_hid_layers=2, residual_flag=True)
    ensemble.append(net)
    mlp_optimizers.append(torch.optim.Adam(params=net.parameters(), lr=1e-3))
```

```
[ ]: # train each ensemble component
for idx, net in enumerate(ensemble):

    # training loop for one NN
    bar = trange(3000)
    for epoch in bar:
        mlp_optimizers[idx].zero_grad()
        loss = mlp_criterion(y_train, net(x_train))
        loss.backward()
        mlp_optimizers[idx].step()

    0%|          | 0/3000 [00:00<?, ?it/s]
    0%|          | 0/3000 [00:00<?, ?it/s]
```

Evaluate

Same as before, let's investigate how our Deep Ensemble performs on the entire data domain of our input variable x .

```
[ ]: # compute predictions for each network
y_preds = []
for idx, net in enumerate(ensemble):
    y_preds.append( net(x_test).clone().detach().numpy() )
y_preds = np.array(y_preds)
```

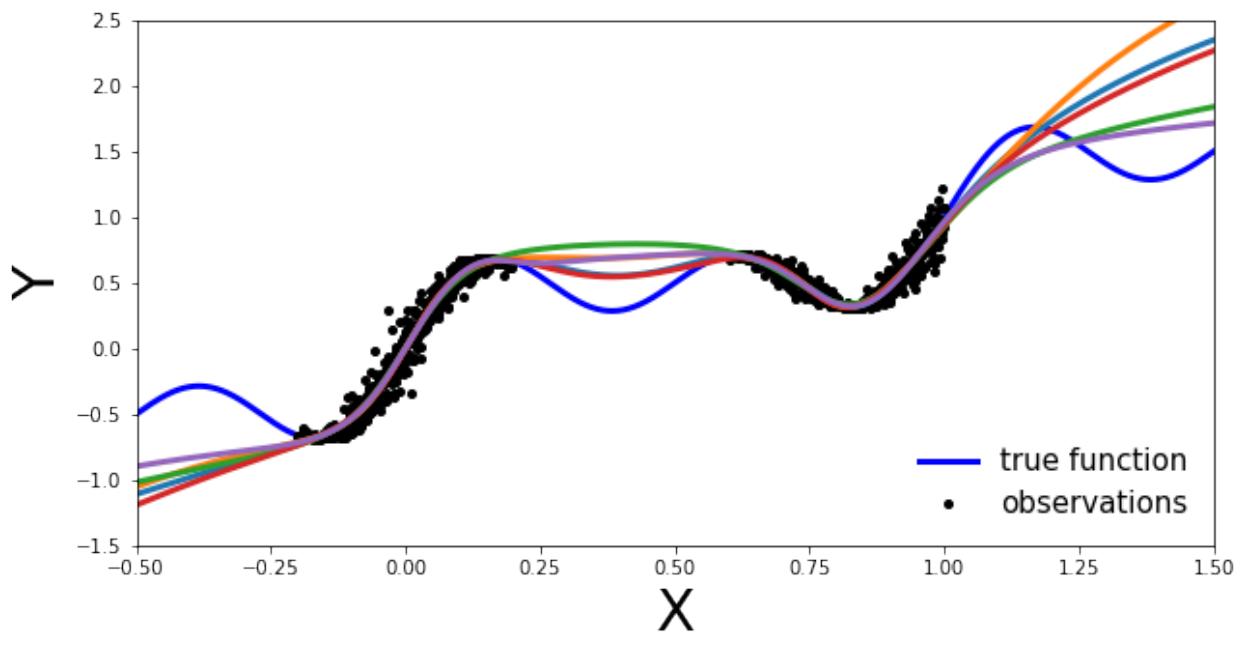
Plot each ensemble member's predictive function.

```
[ ]: fig, ax = plt.subplots(figsize=(10, 5))
plt.xlim(xlims)
plt.ylim(ylims)
plt.xlabel("X", fontsize=30)
plt.ylabel("Y", fontsize=30)

ax.plot(x_true, y_true, 'b-', linewidth=3, label="true function")
ax.plot(x, y, 'ko', markersize=4, label="observations")

# plot each network's function
for idx in range(ensemble_size):
    ax.plot(x_test, y_preds[idx], '-.', linewidth=3)

plt.legend(loc=4, fontsize=15, frameon=False);
```



In this plot the benefit of an ensemble approach is not immediately clear. Still on the regions outside the training data each of the trained NN is inaccurate. So what is the benefit you might ask.

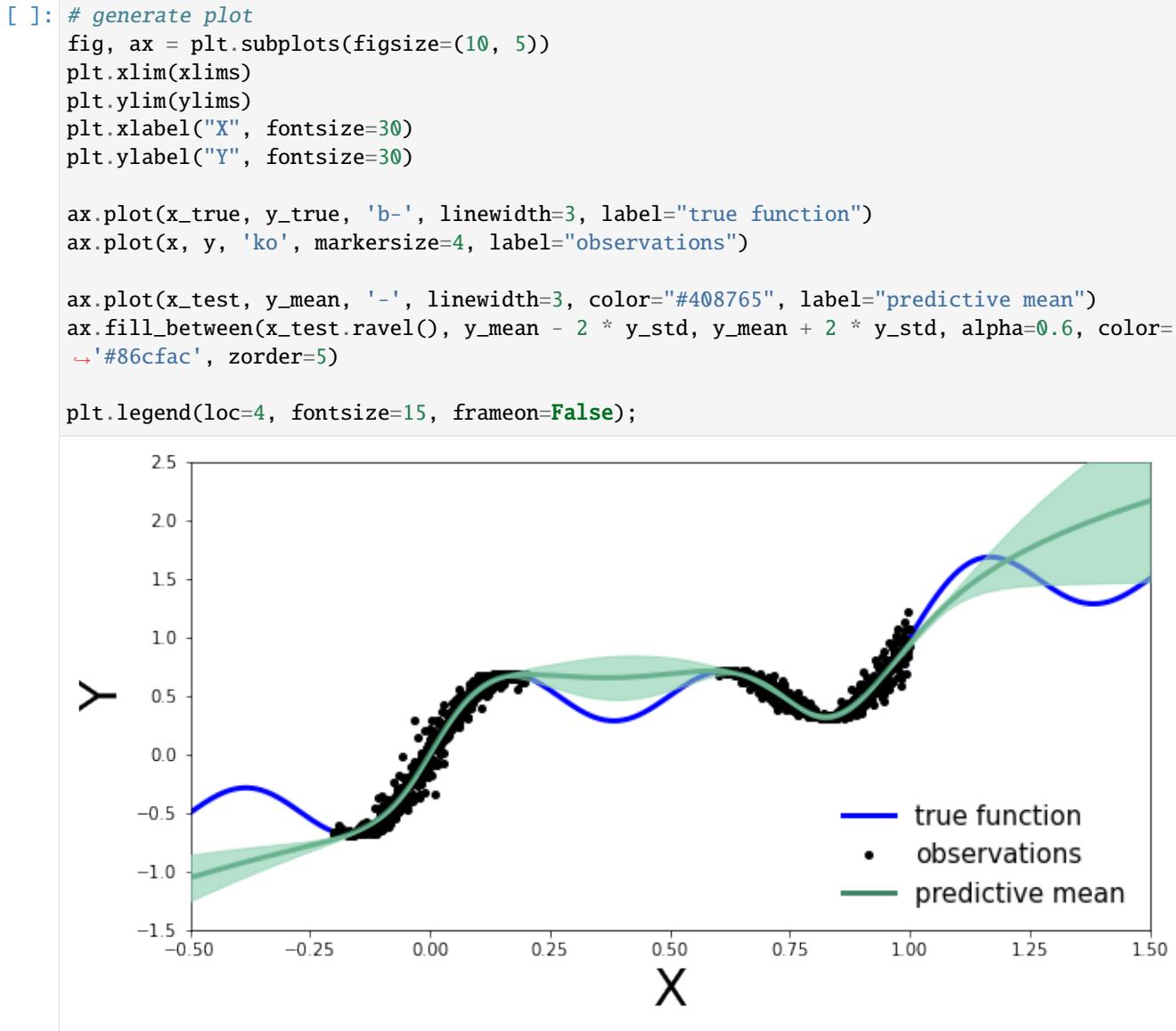
Well let's plot the above in a slightly different way: let's visualize the ensemble's **uncertainty bands**.

From a Bayesian perspective we want to quantify the model's uncertainty on its prediction. This is done via the marginal $p(y|x, D)$, which can be computed as:

$$p(y|x, D) = \int_{\theta} p(y|x, \theta') p(\theta'|D) d\theta'$$

In practice, for Deep Ensembles we approximate the above by computing the mean and standard deviation across the ensemble. Meaning $p(\theta|D)$ represents the parameters of one of the trained models, $\theta_i p(\theta|D)$, which we then use to compute $y_i = f(x, \theta_i)$, representing $p(y|x, \theta')$.

```
[ ]: # compute mean and standard deviation
y_mean = y_preds.mean(axis=0)
y_std = y_preds.std(axis=0)
```



Now we see the benefit of a Bayesian approach. Outside the training region we not only have the point estimate, but also model's uncertainty about it's prediction.

4.38.5 Monte Carlo Dropout

First we create our MC-Dropout Network. As you can see in the code below, creating a dropout network is extremely simple:

we can reuse our existing network architecture, the only alteration is that during the forward pass we randomly *switch off* (zero) some of the elements of the input tensor

The Bayesian interpretation of MC-Dropout is that we can see each dropout configuration as a different sample from the approximate posterior distribution $\theta_i q(\theta|D)$.

Training

```
[ ]: # define MLP with dropout (probability of 0.5)
dropout_mlp = MLP(hid_dim=30, n_hid_layers=2, residual_flag=True, dropout_flag=True,
                   dropout_prob=0.5)

[ ]: # training
mlp_optimizer = torch.optim.Adam(params=dropout_mlp.parameters(), lr=1e-3)
mlp_criterion = nn.MSELoss()

bar = trange(3000)
for epoch in bar:
    mlp_optimizer.zero_grad()
    loss = mlp_criterion(y_train, dropout_mlp(x_train))
    bar.set_postfix(loss=f'loss / {x_train.shape[0]}:{loss:.3f}')
    loss.backward()
    mlp_optimizer.step()

0% | 0/3000 [00:00<?, ?it/s]
```

Evaluate

Similarly to Deep Ensembles, we pass the test data multiple times through the MC-Dropout network. We do so to obtain y_i at the different parameter settings, θ_i of the network, $y_i = f(x, \theta_i)$, governed by the dropout mask.

This is the main difference compared to dropout implementation in a deterministic NN where it serves as a regularization term. In normal dropout application during test time the dropout is **not** applied. Meaning that all connections are present, but the weights are **adjusted**

```
[ ]: n_mc_samples = 100
y_preds = []

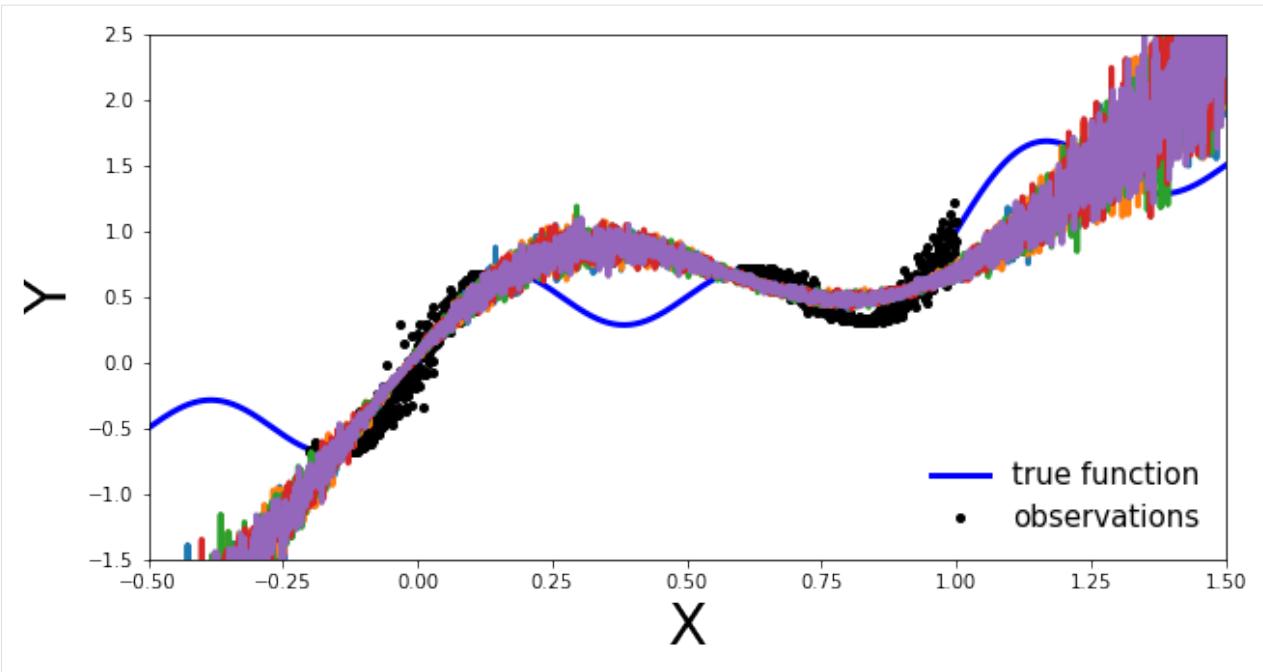
# compute predictions, resampling dropout mask for each forward pass
for idx in range(n_mc_samples):
    y_preds.append(dropout_mlp(x_test).clone().detach().numpy())
y_preds = np.array(y_preds)

[ ]: # generate plot of each dropout (sub)-model
fig, ax = plt.subplots(figsize=(10, 5))
plt.xlim(xlims)
plt.ylim(ylims)
plt.xlabel("X", fontsize=30)
plt.ylabel("Y", fontsize=30)

ax.plot(x_true, y_true, 'b-', linewidth=3, label="true function")
ax.plot(x, y, 'ko', markersize=4, label="observations")

# plot each network's function
for idx in range(ensemble_size):
    ax.plot(x_test, y_preds[idx], 'r--', linewidth=3)

plt.legend(loc=4, fontsize=15, frameon=False);
```



In the above plot each colored line (apart from blue) represents a different parametrization, θ_i , of our MC-Dropout Network.

Likewise to the Deep Ensemble Network, we can also compute the MC-dropout's **uncertainty bands**.

The approach in practice is the same as before: we compute the mean and standard deviation across each dropout mask, which corresponds to the marginal estimation we discussed earlier.

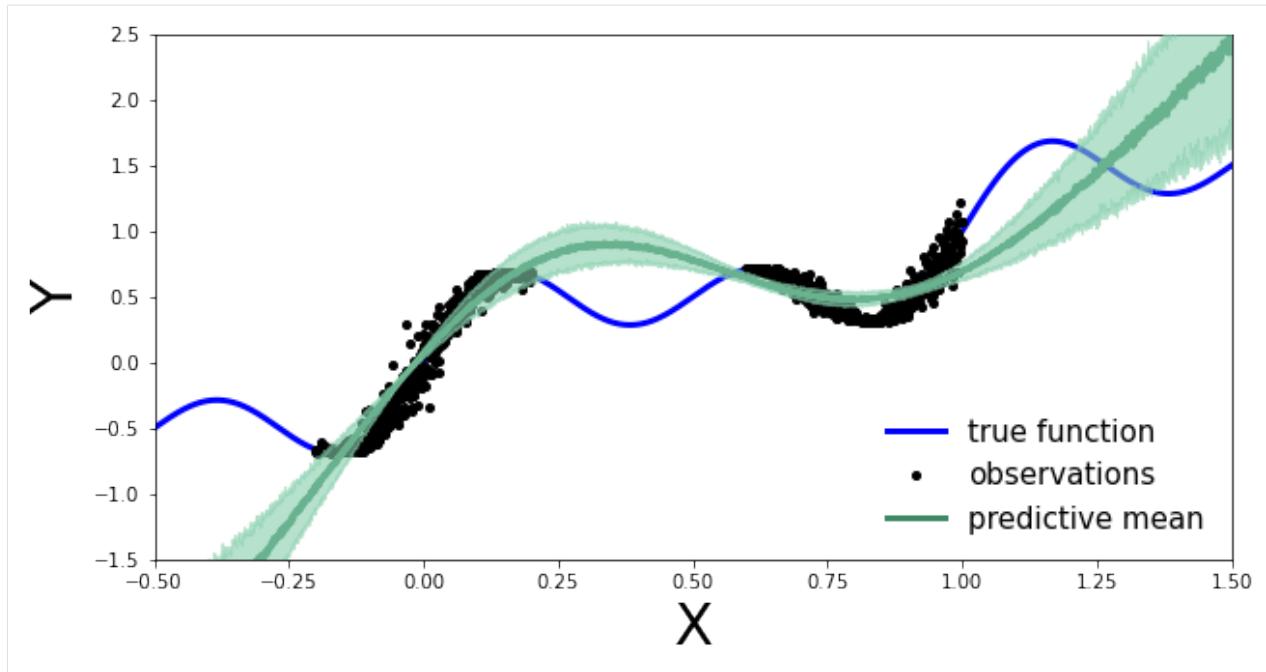
```
[ ]: # compute mean and standard deviation
y_mean = y_preds.mean(axis=0)
y_std = y_preds.std(axis=0)

[ ]: # generate plot
fig, ax = plt.subplots(figsize=(10, 5))
plt.xlim(xlims)
plt.ylim(ylims)
plt.xlabel("X", fontsize=30)
plt.ylabel("Y", fontsize=30)

ax.plot(x_true, y_true, 'b-', linewidth=3, label="true function")
ax.plot(x, y, 'ko', markersize=4, label="observations")

ax.plot(x_test, y_mean, '-', linewidth=3, color="#408765", label="predictive mean")
ax.fill_between(x_test.ravel(), y_mean - 2 * y_std, y_mean + 2 * y_std, alpha=0.6, color="#86cfac", zorder=5)

plt.legend(loc=4, fontsize=15, frameon=False);
```



In the same way as Deep Ensembles, MC-Dropout allows us to have an uncertainty estimate next to our point wise predictions. However, for the given use-case this has come with the cost of an overall drop in the model's performance on the training regions. We observe this because at every pass through our network we randomly choose which nodes to keep, so one could argue that we hinder the networks optimal performance.

Exercise: Detecting Distribution Shift on MNIST

In this exercise we will compare Bayesian NNs with deterministic NN on distribution shift detection task. To do this, we'll monitor the predictive entropy as the distribution gradually shifts. A model with better uncertainty quantification should become less certain—that is, have a more entropic predictive distribution—as the input distribution shifts. Mathematically, our quantity of interest is:

$$\mathbb{H}[y|x^*, D] = - \sum_y p(y|x^*, D) \log p(y|x^*, D)$$

where $p(y|x^*, D)$ is the predictive distribution:

The goal is to essentially replicate Figure 1 from the paper ‘Multiplicative Normalizing Flows for Variational Bayesian Neural Networks’.

We will be using the MNIST dataset, a set of 70,000 hand-written digit images, and we will generate a gradual distribution shift on the dataset by rotating the images. As such, the final plot will depict the change in the entropy of the predictive distribution (y-axis) as degree of rotation increases (x-axis). The paper above shows the result for one image. We, on the other hand, will average over multiple images to make a better comparison between models.

First, let's load MNIST...

```
[ ]: from torchvision import datasets

# download MNIST
mnist_train = datasets.MNIST('../data', train=True, download=True)
mnist_test = datasets.MNIST('../data', train=False, download=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ../data/MNIST/
˓→raw/train-images-idx3-ubyte.gz
 0%|          | 0/9912422 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/train-images-idx3-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ../data/MNIST/
˓→raw/train-labels-idx1-ubyte.gz
 0%|          | 0/28881 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/train-labels-idx1-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ../data/MNIST/
˓→raw/t10k-images-idx3-ubyte.gz
 0%|          | 0/1648877 [00:00<?, ?it/s]

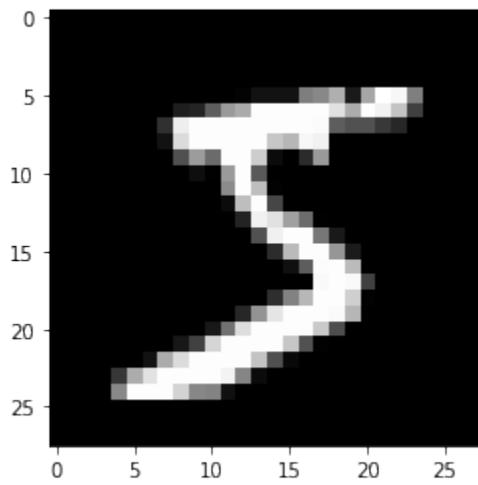
Extracting ../data/MNIST/raw/t10k-images-idx3-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ../data/MNIST/
˓→raw/t10k-labels-idx1-ubyte.gz
 0%|          | 0/4542 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/MNIST/raw
```

Visualize an image...

```
[ ]: img_idx = 0
plt.imshow(mnist_train.data[img_idx], cmap='gray', vmin=0, vmax=255);
```

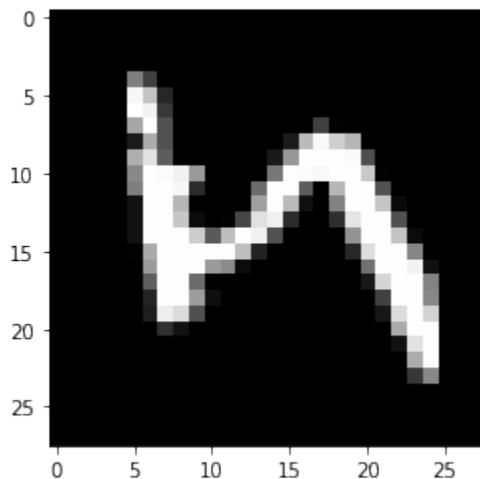


We'll use rotation to simulate a smooth shift. Here's how you can rotate a given image...

```
[ ]: import torchvision.transforms.functional as TF

rotation_angle = 90
rotated_image = TF.rotate(mnist_train.data[img_idx][None], rotation_angle)[0]

plt.imshow(rotated_image, cmap='gray', vmin=0, vmax=255);
```



Let's setup the training data...

```
[ ]: x_train = mnist_train.data.reshape(60000, 28*28).float()
y_train = mnist_train.targets

x_test = mnist_test.data.reshape(10000, 28*28).float()
y_test = mnist_test.targets
```

Now that we have the data, let's start training neural networks.

4.38.6 Deterministic Network

We will reuse our MLP network architecture but increase the hidden layer dimensionality as well as the number of hidden layers:

```
[ ]: mlp = MLP(in_dim=784, out_dim=10, hid_dim=200, n_hid_layers=3, residual_flag=True,
             dropout_flag=False)
```

Training

```
[ ]: # training
mlp_optimizer = torch.optim.Adam(params=mlp.parameters(), lr=1e-4)
mlp_criterion = nn.CrossEntropyLoss()
batch_size = 250

bar = trange(30)
for epoch in bar:
    for batch_idx in range(int(x_train.shape[0] / batch_size)):
        batch_low, batch_high = batch_idx * batch_size, (batch_idx+1) * batch_size
```

(continues on next page)

(continued from previous page)

```

mlp_optimizer.zero_grad()
loss = mlp_criterion(target=y_train[batch_low:batch_high], input=mlp(x_train[batch_
low:batch_high]))
bar.set_postfix(loss=f'{loss / batch_size:.3f}') #x.shape[0]
loss.backward()
mlp_optimizer.step()

0%|          | 0/30 [00:00<?, ?it/s]

```

Test

```

[ ]: def accuracy(targets, predictions):
    return (targets == predictions).sum() / targets.shape[0]

[ ]: # get test accuracy, to make sure we have a satisfactory model
mlp.eval()
test_preds = mlp(x_test).argmax(1)
acc = accuracy(y_test, test_preds)

print("Test accuracy is %.2f%%" %(acc.item() * 100))
Test accuracy is 96.49%

```

4.38.7 Rotating the images

Now let's compute predictive entropy on some rotated images...

First we will generate the rotated images with an increasing rotation angle from the test images:

```

[ ]: # use angles from 0 to 90 degrees
rotation_angles = [3 * i for i in range(0, 31)]
n_test_images = 100

rotated_images = []
for angle in rotation_angles:
    rotated_images.append(TF.rotate(mnist_test.data[:n_test_images].float()[None],_
    angle)[0].reshape(n_test_images, 28*28))

```

Evaluate the trained MLP on the rotated images:

```

[ ]: softmax = torch.nn.functional.softmax

[ ]: deter_pred_means = []
for image in rotated_images:
    deter_pred_means.append(softmax(mlp(image), dim=1))

[ ]: def entropy(p):
    return (-p * np.log(p)).sum(axis=1)

[ ]: # generate plot

```

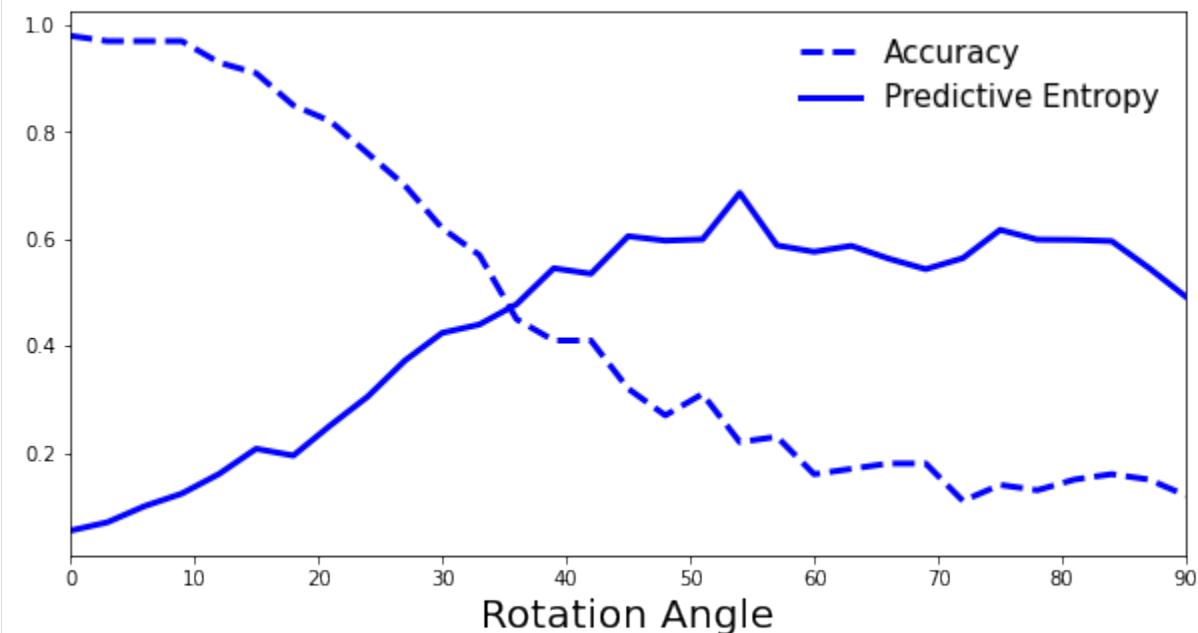
(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(figsize=(10, 5))
plt.xlim([0, 90])
plt.xlabel("Rotation Angle", fontsize=20)

ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in
    deter_pred_means], 'b--', linewidth=3, label="Accuracy")
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in deter_pred_
    means], 'b-', linewidth=3, label="Predictive Entropy")

plt.legend(loc=1, fontsize=15, frameon=False);
```



In the above plot we can see how well our deterministic MLP performs on the test images as the rotation angle is increased (increasing distribution shift). As expected, with a higher rotation angle model's accuracy drops and its predictive entropy increases, as the network class predictions are more uniformly distributed (recall the predictive entropy equation)

4.38.8 Monte Carlo Dropout Network

Let's create our Dropout Network. We keep the network depth and hidden layer size the same as for the MLP for a fair model comparison

```
[ ]: dropout_mlp = MLP(in_dim=784, out_dim=10, hid_dim=200, n_hid_layers=3, residual_
    flag=True, dropout_flag=True)
```

Training

```
[ ]: # training
mlp_optimizer = torch.optim.Adam(params=dropout_mlp.parameters(), lr=1e-4)
mlp_criterion = nn.CrossEntropyLoss()
batch_size = 250
```

(continues on next page)

(continued from previous page)

```

bar = trange(30)
for epoch in bar:
    for batch_idx in range(int(x_train.shape[0] / batch_size)):
        batch_low, batch_high = batch_idx * batch_size, (batch_idx+1) * batch_size
        mlp_optimizer.zero_grad()
        loss = mlp_criterion(target=y_train[batch_low:batch_high], input=dropout_mlp(x_train[batch_low:batch_high]))
        bar.set_postfix(loss=f'loss / batch_size:{loss / batch_size:.3f}')
        loss.backward()
        mlp_optimizer.step()

0% | 0/30 [00:00<?, ?it/s]

```

Test

```

[ ]: # get test accuracy
dropout_mlp.eval()
test_preds = dropout_mlp(x_test).argmax(1)
acc = accuracy(y_test, test_preds)

print("Test accuracy is %.2f%%" %(acc.item() * 100))
Test accuracy is 96.57%

```

Evaluate on rotated images

```

[ ]: n_mc_samples = 100
dropout_mlp.train() #we set the model to train so the dropout layer is 'active'

dropout_pred_means = []
for image in rotated_images:
    # compute predictions, resampling dropout mask for each forward pass
    y_preds = torch.zeros((n_test_images, 10))
    for idx in range(n_mc_samples):
        y_preds += softmax(dropout_mlp(image), dim=1)
    dropout_pred_means.append( y_preds / n_mc_samples )

[ ]: # generate plot
fig, ax = plt.subplots(figsize=(10, 5))
plt.xlim([0, 90])
plt.ylim([0, 1.2])
plt.xlabel("Rotation Angle", fontsize=20)

# plot deterministic results
ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in deter_pred_means], 'b--', linewidth=3, label="Acc, Deter.");
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in deter_pred_means], 'b-', linewidth=3, label="Ent, Deter.");

# plot MC dropout results
ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in dropout_pred_means], 'r--', linewidth=3, label="Acc, MC Dropout")

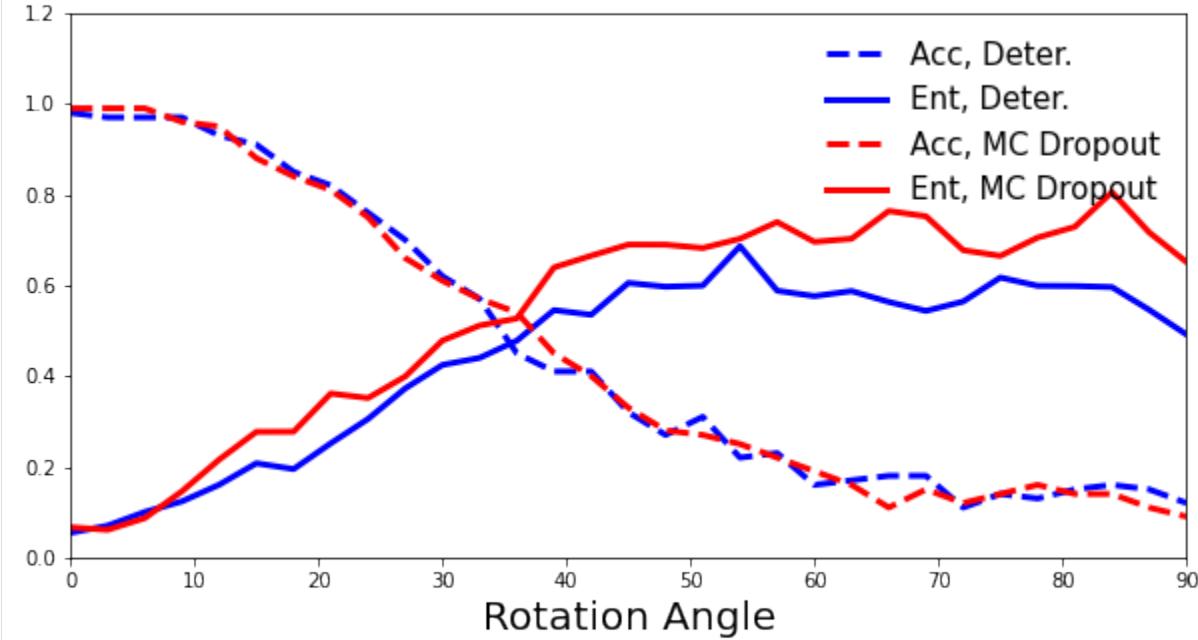
```

(continues on next page)

(continued from previous page)

```
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in dropout_pred_
means], 'r-', linewidth=3, label="Ent, MC Dropout")

plt.legend(loc=1, fontsize=15, frameon=False);
```



The accuracy is on par between the MC-Dropout and MLP. However, now we can see that the MC-Dropout has a better uncertainty quantification as it's predictive entropy is higher. What this means is that overall both model prediction uncertainty increases as the distribution shift increases, however, as MC-dropout computes a model average it is more sensitive (more uncertain) about the predictions with a higher rotation angle.

4.38.9 Deep Ensemble

Now let's investigate Deep Ensemble performance. We will use the exact same network hyperparameters as for the MLP:

```
[ ]: # define the size of the ensemble: number of NN to train
ensemble_size = 5
ensemble = []
mlp_optimizers = []

# initialize the ensemble
for _ in range(ensemble_size):
    net = MLP(in_dim=784, out_dim=10, hid_dim=200, n_hid_layers=3, residual_flag=True,_
    dropout_flag=False)
    ensemble.append(net)
    mlp_optimizers.append(torch.optim.Adam(params=net.parameters(), lr=1e-4))
```

Training

```
[ ]: # train each ensemble component
batch_size = 250
mlp_criterion = nn.CrossEntropyLoss()

for idx, net in enumerate(ensemble):
    # training loop for one NN
    net = ensemble[idx]
    bar = trange(30)
    for epoch in bar:
        for batch_idx in range(int(x_train.shape[0] / batch_size)):
            batch_low, batch_high = batch_idx * batch_size, (batch_idx+1) * batch_size
            mlp_optimizers[idx].zero_grad()
            loss = mlp_criterion(target=y_train[batch_low:batch_high], input=net(x_train[batch_low:batch_high]))
            bar.set_postfix(loss=f'{loss / batch_size:.3f}')
            loss.backward()
            mlp_optimizers[idx].step()

    0% | 0/30 [00:00<?, ?it/s]
    0% | 0/30 [00:00<?, ?it/s]
```

Test

```
[ ]: # get test accuracy
# compute predictions for each network
test_preds = []
for idx, net in enumerate(ensemble):
    net.eval()
    test_preds.append(net(x_test).argmax(1))

test_preds = torch.stack(test_preds, dim=0)
test_preds = torch.sum(test_preds, dim=0)*1/ensemble_size #we average the predictions over the ensemble

# compute the accuracy
acc = accuracy(y_test, test_preds.int())

print("Test accuracy is %.2f%%" %(acc.item() * 100))
Test accuracy is 93.88%
```

Evaluate on rotated images

```
[ ]: ensemble_pred_means = []
for image in rotated_images:
    # compute predictions for each ensemble network
    y_preds = torch.zeros((n_test_images, 10))
    for idx, net in enumerate(ensemble):
```

(continues on next page)

(continued from previous page)

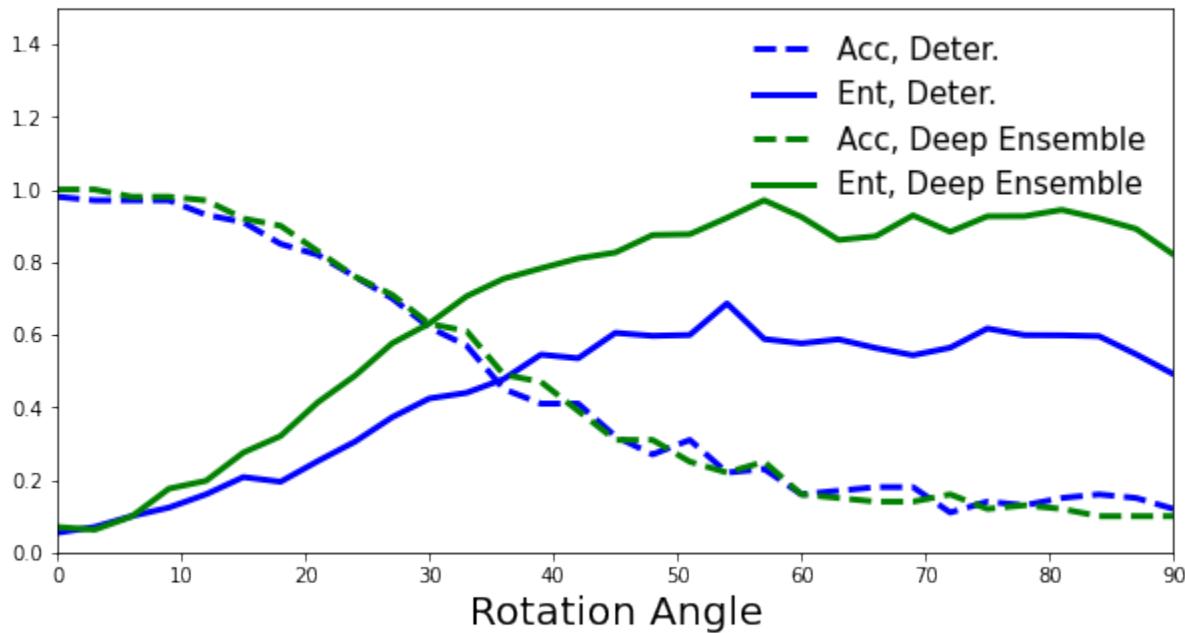
```
y_preds += softmax(net(image), dim=1)
ensemble_pred_means.append( y_preds / ensemble_size )
```

```
[ ]: # generate plot
fig, ax = plt.subplots(figsize=(10, 5))
plt.xlim([0, 90])
plt.ylim([0, 1.5])
plt.xlabel("Rotation Angle", fontsize=20)

# plot deterministic results
ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in deter_pred_means], 'b--', linewidth=3, label="Acc, Deter.");
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in deter_pred_means], 'b-', linewidth=3, label="Ent, Deter.");

# plot Ensemble results
ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in ensemble_pred_means], 'g--', linewidth=3, label="Acc, Deep Ensemble")
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in ensemble_pred_means], 'g-', linewidth=3, label="Ent, Deep Ensemble")

plt.legend(loc=1, fontsize=15, frameon=False);
```



The accuracy of the Deep Ensemble is initially lower than of our deterministic MLP. However, this is accounted by the fact that the Deep Ensemble is better in uncertainty quantification as it's predictive entropy is higher. How to interpret this? Well, in Deep Ensemble we have trained multiple deterministic MLPs. As the rotation angle increases the point-estimates of each model become more varied, hence the average of these point estimates, $p(y|x^*, D)$, has a more uniform distribution across the classes. As such, the resulting predictive entropy is higher, as the log of $p(y|x^*, D)$ is more negative for smaller values (recall `log` function).

4.38.10 Bayesian Neural Network

We will train a *truly* Bayesian Neural Netowrk with SVI

First, lets install the pyro package:

```
[ ]: !pip install pyro-ppl
Collecting pyro-ppl
  Downloading pyro_ppl-1.8.1-py3-none-any.whl (718 kB)
    || 718 kB 5.3 MB/s
Collecting pyro-api>=0.1.1
  Downloading pyro_api-0.1.2-py3-none-any.whl (11 kB)
Requirement already satisfied: tqdm>=4.36 in /usr/local/lib/python3.7/dist-packages_
  ↪(from pyro-ppl) (4.64.0)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.7/dist-
  ↪packages (from pyro-ppl) (3.3.0)
Collecting torch>=1.11.0
  Downloading torch-1.11.0-cp37-cp37m-manylinux1_x86_64.whl (750.6 MB)
    || 750.6 MB 12 kB/s
Requirement already satisfied: numpy>=1.7 in /usr/local/lib/python3.7/dist-packages_
  ↪(from pyro-ppl) (1.21.6)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-
  ↪packages (from torch>=1.11.0->pyro-ppl) (4.1.1)
Installing collected packages: torch, pyro-api, pyro-ppl
  Attempting uninstall: torch
    Found existing installation: torch 1.10.0+cu111
    Uninstalling torch-1.10.0+cu111:
      Successfully uninstalled torch-1.10.0+cu111
ERROR: pip's dependency resolver does not currently take into account all the packages_
  ↪that are installed. This behaviour is the source of the following dependency conflicts.
torchvision 0.11.1+cu111 requires torch==1.10.0, but you have torch 1.11.0 which is_
  ↪incompatible.
torchtext 0.11.0 requires torch==1.10.0, but you have torch 1.11.0 which is incompatible.
torchaudio 0.10.0+cu111 requires torch==1.10.0, but you have torch 1.11.0 which is_
  ↪incompatible.
Successfully installed pyro-api-0.1.2 pyro-ppl-1.8.1 torch-1.11.0
```

Data type cannot be displayed: application/vnd.colab-display-data+json

```
[ ]: import pyro
import pyro.distributions as dist
from pyro.nn import PyroModule, PyroSample
from pyro.infer import Predictive
from pyro.infer import SVI, Trace_ELBO
from pyro.infer.autoguide import AutoDiagonalNormal
from pyro.distributions import Normal, Categorical
from torch.nn.functional import softmax
from tqdm.auto import trange, tqdm
```

Now let's build our BNN!

As a backbone use the MLP architecture introduced in the beginning of the notebook. However, because we will implement a custom *guide()*, define every layer explicitly.

```
[ ]: class My_MLP(nn.Module):

    def __init__(self, in_dim=784, out_dim=10, hid_dim=200):
        super().__init__()
        assert in_dim > 0
        assert out_dim > 0
        assert hid_dim > 0
        self.activation = nn.Tanh() # could also be nn.ReLU()

        #3 hidden layers
        self.fc1 = nn.Linear(in_dim, hid_dim)
        self.fc2 = nn.Linear(hid_dim, hid_dim)
        self.fc3 = nn.Linear(hid_dim, hid_dim)
        self.out = nn.Linear(hid_dim, out_dim)

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.activation(self.fc3(x))

        #hid --> output prob
        yhat = self.out(x).squeeze()
        return yhat
```

Initialize the network. You will have to access it's layers in your model and guide functions

```
[ ]: net = My_MLP()

[ ]: #confirm your layer names
for name, _ in net.named_parameters():
    print(name)

fc1.weight
fc1.bias
fc2.weight
fc2.bias
fc3.weight
fc3.bias
out.weight
out.bias
```

Define the model:

Probabilistic models in Pyro are specified as *model()* functions. This function defines how the output data is generated. Within the *model()* function, first, the pyro module *random_module()* converts the paramters of our NN into random variables that have prior probability distributions. Second, in pyro *sample* we define that the output of the network is categorical, while the pyro *plate* allows us to vectorize this function for computational efficiency.

Hint: remember we are doing a classification instead of regression!

We will ‘cheat’ a little: to speed up training and limit a bit more the number of paramters we need to optimize, we will implement a BNN where only the **last layer** is Bayesian!

```
[ ]: #if you want to try later a full BNN, you can uncomment the other lines
def model(x_data, y_data):

    #weight and bias priors

    # fc1w_prior = Normal(loc=torch.zeros_like(net.fc1.weight), scale=torch.ones_
    ↪like(net.fc1.weight)).to_event(2)
    # fc1b_prior = Normal(loc=torch.zeros_like(net.fc1.bias), scale=torch.ones_like(net.
    ↪fc1.bias)).to_event(1)

    # fc2w_prior = Normal(loc=torch.zeros_like(net.fc2.weight), scale=torch.ones_
    ↪like(net.fc2.weight)).to_event(2)
    # fc2b_prior = Normal(loc=torch.zeros_like(net.fc2.bias), scale=torch.ones_like(net.
    ↪fc2.bias)).to_event(1)

    # fc3w_prior = Normal(loc=torch.zeros_like(net.fc3.weight), scale=torch.ones_
    ↪like(net.fc3.weight)).to_event(2)
    # fc3b_prior = Normal(loc=torch.zeros_like(net.fc3.bias), scale=torch.ones_like(net.
    ↪fc3.bias)).to_event(1)

    outw_prior = Normal(loc=torch.zeros_like(net.out.weight), scale=torch.ones_like(net.
    ↪out.weight)).to_event(2)
    outb_prior = Normal(loc=torch.zeros_like(net.out.bias), scale=torch.ones_like(net.
    ↪out.bias)).to_event(1)

    priors = {
        # 'fc1.weight': fc1w_prior, 'fc1.bias': fc1b_prior,
        # 'fc2.weight': fc2w_prior, 'fc2.bias': fc2b_prior,
        # 'fc3.weight': fc3w_prior, 'fc3.bias': fc3b_prior,
        'out.weight': outw_prior, 'out.bias': outb_prior}

    # lift module parameters to random variables sampled from the priors
    lifted_module = pyro.random_module("module", net, priors)
    # sample a regressor (which also samples w and b)
    lifted_reg_model = lifted_module()

    with pyro.plate("data", x_data.shape[0]):
        yhat = softmax(lifted_reg_model(x_data), dim=1)
        obs = pyro.sample("obs", dist.Categorical(yhat), obs=y_data)
```

implement the guide(), *variational distribution*:

the guide allows us to initialise a well behaved distribution which later we can optimize to approximate the true posterior

```
[ ]: softplus = torch.nn.Softplus()

#similarly here, if you want a full BNN uncomment the other layers
def my_guide(x_data, y_data):

    # # First layer weight distribution priors
    # fc0w_mu_param = pyro.param("fc0w_mu", torch.randn_like(net.fc1.weight))
```

(continues on next page)

(continued from previous page)

```

# fc0w_sigma_param = softplus(pyro.param("fc0w_sigma", torch.randn_like(net.fc1.
˓→weight)))
# fc0w_prior = Normal(loc=fc0w_mu_param, scale=fc0w_sigma_param).to_event(2)
# # First layer bias distribution priors
# fc0b_mu_param = pyro.param("fc0b_mu", torch.randn_like(net.fc1.bias))
# fc0b_sigma_param = softplus(pyro.param("fc0b_sigma", torch.randn_like(net.fc1.
˓→bias)))
# fc0b_prior = Normal(loc=fc0b_mu_param, scale=fc0b_sigma_param).to_event(1)

# # Second layer weight distribution priors
# fc1w_mu_param = pyro.param("fc1w_mu", torch.randn_like(net.fc2.weight))
# fc1w_sigma_param = softplus(pyro.param("fc1w_sigma", torch.randn_like(net.fc2.
˓→weight)))
# fc1w_prior = Normal(loc=fc1w_mu_param, scale=fc1w_sigma_param).to_event(2)
# # Second layer bias distribution priors
# fc1b_mu_param = pyro.param("fc1b_mu", torch.randn_like(net.fc2.bias))
# fc1b_sigma_param = softplus(pyro.param("fc1b_sigma", torch.randn_like(net.fc2.
˓→bias)))
# fc1b_prior = Normal(loc=fc1b_mu_param, scale=fc1b_sigma_param).to_event(1)

# # Third layer weight distribution priors
# fc2w_mu_param = pyro.param("fc2w_mu", torch.randn_like(net.fc3.weight))
# fc2w_sigma_param = softplus(pyro.param("fc2w_sigma", torch.randn_like(net.fc3.
˓→weight)))
# fc2w_prior = Normal(loc=fc2w_mu_param, scale=fc2w_sigma_param).to_event(2)
# # Third layer bias distribution priors
# fc2b_mu_param = pyro.param("fc2b_mu", torch.randn_like(net.fc3.bias))
# fc2b_sigma_param = softplus(pyro.param("fc2b_sigma", torch.randn_like(net.fc3.
˓→bias)))
# fc2b_prior = Normal(loc=fc2b_mu_param, scale=fc2b_sigma_param).to_event(1)

# Output layer weight distribution priors
outw_mu_param = pyro.param("outw_mu", torch.randn_like(net.out.weight))
outw_sigma_param = softplus(pyro.param("outw_sigma", torch.randn_like(net.out.
˓→weight)))
outw_prior = Normal(loc=outw_mu_param, scale=outw_sigma_param).to_event(2)
# Output layer bias distribution priors
outb_mu_param = pyro.param("outb_mu", torch.randn_like(net.out.bias))
outb_sigma_param = softplus(pyro.param("outb_sigma", torch.randn_like(net.out.bias)))
outb_prior = Normal(loc=outb_mu_param, scale=outb_sigma_param).to_event(1)

priors = {
    # 'fc1.weight': fc0w_prior, 'fc1.bias': fc0b_prior,
    # 'fc2.weight': fc1w_prior, 'fc2.bias': fc1b_prior,
    # 'fc3.weight': fc2w_prior, 'fc3.bias': fc2b_prior,
    'out.weight': outw_prior, 'out.bias': outb_prior}

lifted_module = pyro.random_module("module", net, priors)

return lifted_module()

```

Initialize the stochastic variational inference (SVI)

```
[ ]: adam = pyro.optim.Adam({"lr": 1e-3})
svi = SVI(model, my_guide, adam, loss=Trace_ELBO())
```

Training

```
[ ]: pyro.clear_param_store()
batch_size = 250
bar = trange(30)
for epoch in bar:
    for batch_idx in range(int(x_train.shape[0] / batch_size)):
        batch_low, batch_high = batch_idx * batch_size, (batch_idx+1) * batch_size
        loss = svi.step(x_train[batch_low:batch_high], y_train[batch_low:batch_high])
        bar.set_postfix(loss=f'{loss / batch_size:.3f}')
    0%|          | 0/30 [00:00<?, ?it/s]
/usr/local/lib/python3.7/dist-packages/pyro/primitives.py:495: FutureWarning: The
↳ `random_module` primitive is deprecated, and will be removed in a future release. Use
↳ `pyro.nn.Module` to create Bayesian modules from `torch.nn.Module` instances.
FutureWarning,
```

Test

We are going to use our learned *guide()* function to do predictions. Why? Because the *model()* function knows the **priors** for the weights and biases, **not** the learned posterior. The *guide()* contains the approximate posterior distributions of the parameter values, which we want to use to make the predictions.

```
[ ]: #we sample new sets of weights and parameters 10 times given by num_samples
num_samples = 10
def predict(x):
    sampled_models = [my_guide(None, None) for _ in range(num_samples)]
    yhats = [model(x).data for model in sampled_models]
    mean = torch.mean(torch.stack(yhats), 0)
    return mean

test_preds = predict(x_test).argmax(1)
acc = accuracy(y_test, test_preds)
print("Test accuracy is %.2f%%" %(acc.item() * 100))
/usr/local/lib/python3.7/dist-packages/pyro/primitives.py:495: FutureWarning: The
↳ `random_module` primitive is deprecated, and will be removed in a future release. Use
↳ `pyro.nn.Module` to create Bayesian modules from `torch.nn.Module` instances.
FutureWarning,
```

Test accuracy is 91.63%

Evaluate on rotated images

```
[ ]: num_samples = 50
def predict_probability(x):
    sampled_models = [my_guide(None, None) for _ in range(num_samples)]
    yhats = [softmax(model(x).data, dim=1) for model in sampled_models]
    mean = torch.mean(torch.stack(yhats), 0)
    return mean
```

(continues on next page)

(continued from previous page)

```
bnn_pred_means = []
for image in rotated_images:
    # compute predictions
    bnn_pred_means.append(predict_probability(image))

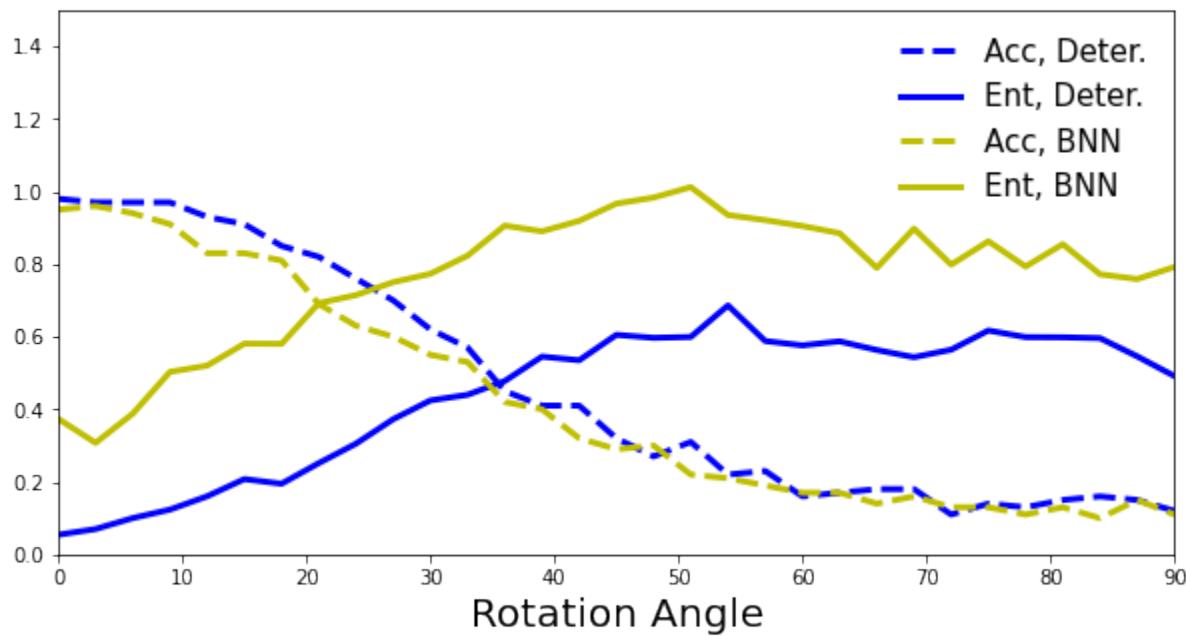
/usr/local/lib/python3.7/dist-packages/pyro/primitives.py:495: FutureWarning: The
↳ `random_module` primitive is deprecated, and will be removed in a future release. Use
↳ `pyro.nn.Module` to create Bayesian modules from `torch.nn.Module` instances.
FutureWarning,
```

```
[ ]: # generate plot
fig, ax = plt.subplots(figsize=(10, 5))
plt.xlim([0, 90])
plt.ylim([0, 1.5])
plt.xlabel("Rotation Angle", fontsize=20)

# plot deterministic results
ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in
    deter_pred_means], 'b--', linewidth=3, label="Acc, Deter.");
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in deter_pred_
    means], 'b-', linewidth=3, label="Ent, Deter.");

# plot Ensemble results
ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in
    bnn_pred_means], 'y--', linewidth=3, label="Acc, BNN")
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in bnn_pred_means],
    'y-', linewidth=3, label="Ent, BNN")

plt.legend(loc=1, fontsize=15, frameon=False);
```



4.38.11 Show entropies for all four models.

Which is the best at detecting the distribution shift?

```
[ ]: # generate plot
fig, ax = plt.subplots(figsize=(10, 5))
plt.xlim([0, 90])
plt.ylim([0, 1.5])
plt.xlabel("Rotation Angle", fontsize=20)

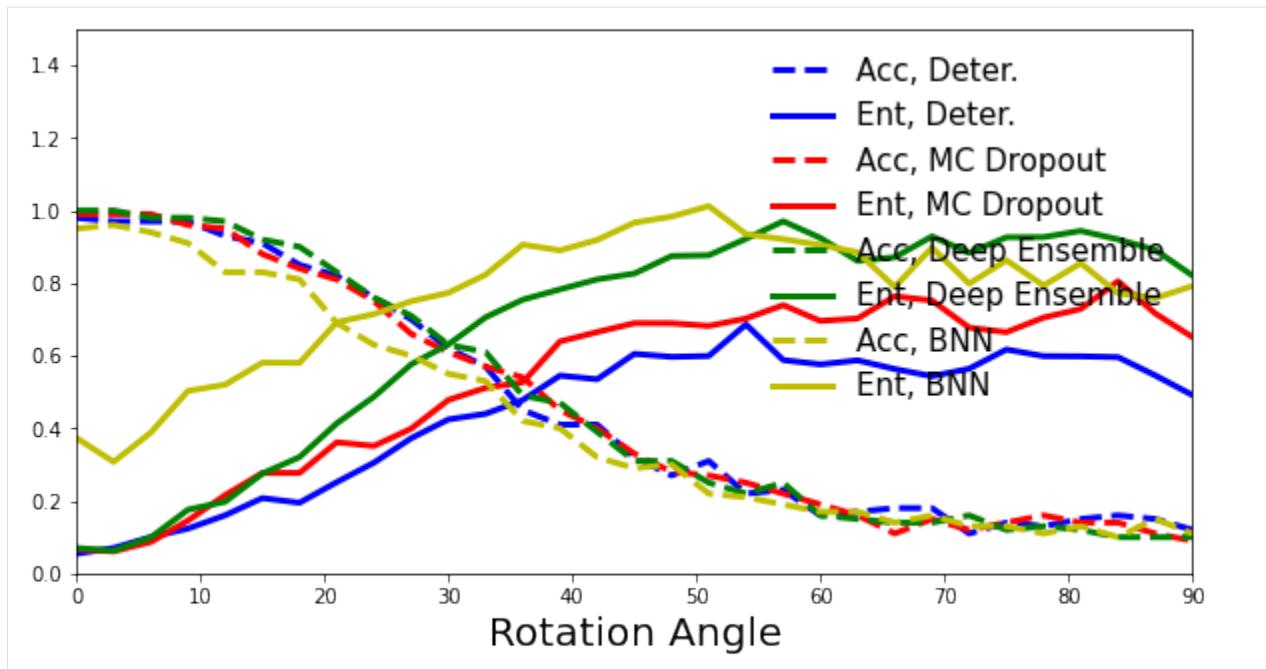
# plot deterministic results
ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in deter_pred_means], 'b--', linewidth=3, label="Acc, Deter.");
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in deter_pred_means], 'b-', linewidth=3, label="Ent, Deter.");

# plot MC dropout results
ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in dropout_pred_means], 'r--', linewidth=3, label="Acc, MC Dropout")
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in dropout_pred_means], 'r-', linewidth=3, label="Ent, MC Dropout")

# plot Ensemble results
ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in ensemble_pred_means], 'g--', linewidth=3, label="Acc, Deep Ensemble")
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in ensemble_pred_means], 'g-', linewidth=3, label="Ent, Deep Ensemble")

# plot BNN results
ax.plot(rotation_angles, [accuracy(y_test[:n_test_images], p.argmax(axis=1)) for p in bnn_pred_means], 'y--', linewidth=3, label="Acc, BNN")
ax.plot(rotation_angles, [np.mean(entropy(p.detach().numpy())) for p in bnn_pred_means], 'y-', linewidth=3, label="Ent, BNN")

plt.legend(loc=1, fontsize=15, frameon=False);
```



4.39 DNN - Tutorial 2 Part I: Physics inspired Machine Learning

Sparse Identification of Nonlinear Dynamical Systems (SINDy)

Filled notebook:

Authors: Leonard Bereska and Ilze Amanda Auzina

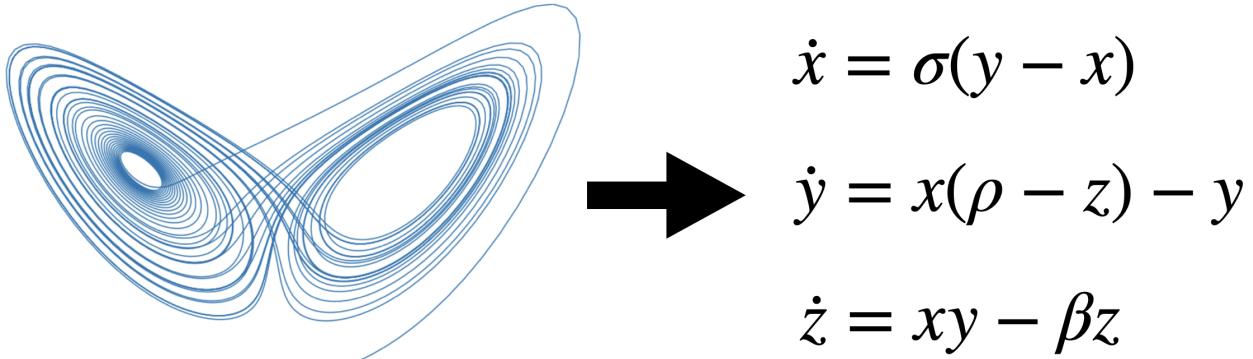
This section provides an overview of the Sparse Identification of Nonlinear Dynamical systems (SINDy) algorithm, which was first proposed in:

Brunton, Steven L., Joshua L. Proctor, and J. Nathan Kutz. 2016. “*Discovering Governing Equations from Data by Sparse Identification of Nonlinear Dynamical Systems.*” Proceedings of the National Academy of Sciences 113 (15): 3932–37. <https://doi.org/10.1073/pnas.1517384113>.

4.39.1 What do we mean by *identifying* nonlinear dynamics from data?

Suppose we have time-series data that originate from a (nonlinear) dynamical system.

- To identify a system means to infer the governing equations of that system based on the data. - That is, to find f for dynamical systems equations $\dot{\mathbf{x}} = f(\mathbf{x})$ (where \mathbf{x} may be vector-valued).



For example for the Lorenz system, we want to learn the **equations** on the *right* from the time-series **data** visualized on the *left*.

4.39.2 Why do we want sparsity?

In this context, by sparse we mean a low number of terms in the governing equations. Sparsity is beneficial, because it is more: 1. **Interpretable**. Crucial, when understanding the variables and their interactions is needed, e.g. for applications with safety-critical guarantees. 2. **Generalizable**. If correct, the equations describe the dynamics accurately beyond the region of state space filled by the training data.

In general, one can think of the models identified by SINDy as equations from physics as opposed to big, opaque, deep neural networks.

4.39.3 SINDy Algorithm

SINDy tries to find the dynamical system f that suits the data $\dot{\mathbf{X}} = f(\mathbf{X})$. This function approximation problem is formulated as linear regression $\dot{\mathbf{X}} = \Theta(\mathbf{X})\Xi$, with coefficients Ξ and a library of regression terms $\Theta(\mathbf{X})$. The algorithm proceeds in three steps:

1. Generate data X from a dynamical system and compute derivatives \dot{X} .
2. Set up library of candidate terms $\Theta(X)$ as functions on X .
3. Sparsely regress the coefficients Ξ , that best describe the data.

1. SINDy assumes a time-series of n -dimensional data points $\mathbf{x} = (x_1, \dots, x_n)$ measured at m time steps t_1, \dots, t_m , we define the data matrix X as and derivatives matrix \dot{X} :

$$X = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \dots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \dots & x_n(t_2) \\ \vdots & \vdots & & \vdots \\ x_1(t_m) & x_2(t_m) & \dots & x_n(t_m) \end{bmatrix}, \quad \dot{X} = \begin{bmatrix} \dot{x}_1(t_1) & \dot{x}_2(t_1) & \dots & \dot{x}_n(t_1) \\ \dot{x}_1(t_2) & \dot{x}_2(t_2) & \dots & \dot{x}_n(t_2) \\ \vdots & \vdots & & \vdots \\ \dot{x}_1(t_m) & \dot{x}_2(t_m) & \dots & \dot{x}_n(t_m) \end{bmatrix}.$$

2. Next, we define the library matrix $\Theta(X)$, the columns of which are a set of basis functions $\{\theta_l\}_{l=1,\dots,L}$ applied to the data:

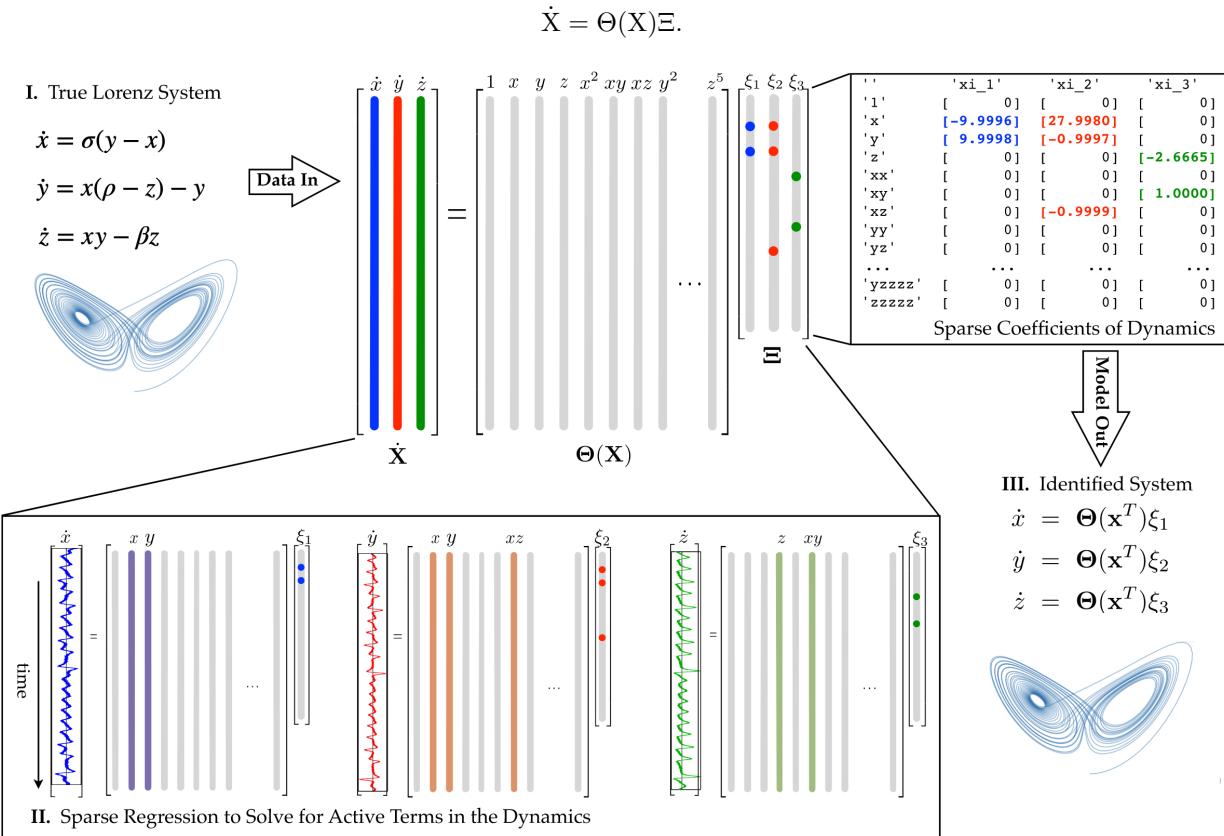
$$\Theta(X) = \begin{bmatrix} \theta_1(X) & \theta_2(X) & \dots & \theta_\ell(X) \end{bmatrix}.$$

Simple examples are e.g. a basis of polynomials $x_1, x_2, x_1^2, x_2^2, x_1x_2, x_1^3, \dots$, such as in the *Taylor expansion*, or trigonometric functions $\sin(x_1), \cos(x_1), \sin^2(2x_1), \dots$, such as in a *Fourier expansion*. But, depending on the problem more complex basis may be appropriate, e.g. Bessel functions.

3. Lastly, we employ a sparse linear regression algorithm (such as e.g. LASSO) to find the coefficients Ξ ,

$$\Xi = \begin{bmatrix} & & \\ \xi_1 & \xi_2 & \dots & \xi_n \\ & & \end{bmatrix},$$

such that



4.39.4 Example: Linear Dynamical System

Suppose we measure a particle trajectory that is governed by the following dynamical system:

$$\frac{d}{dt} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -2x \\ y \end{pmatrix} = \begin{pmatrix} -2 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

With the initial conditions $x_0 = 3$ and $y_0 = \frac{1}{2}$, we construct the data matrix X :

```
[4]: import numpy as np
import pysindy as ps

t = np.linspace(0, 1, 100)
x = 3 * np.exp(-2 * t)
y = 0.5 * np.exp(t)
X = np.stack((x, y), axis=-1)
```

When instantiating a SINDy model, we can choose the differentiation method, the library and the optimizer:

Let's first investigate a Fourier basis as a basis function

```
[18]: model = ps.SINDy(  
    differentiation_method=ps.FiniteDifference(order=2),  
    feature_library=ps.FourierLibrary(),  
    optimizer=ps.STLSQ(threshold=0.2),  
    feature_names=['x', 'y'])  
)  
model.fit(X, t=t)  
  
[18]: SINDy(differentiation_method=FiniteDifference(),  
            feature_library=<pysindy.feature_library.fourier_library.FourierLibrary object at  
            ~0x137889100>,  
            feature_names=['x', 'y'], optimizer=STLSQ(threshold=0.2))
```

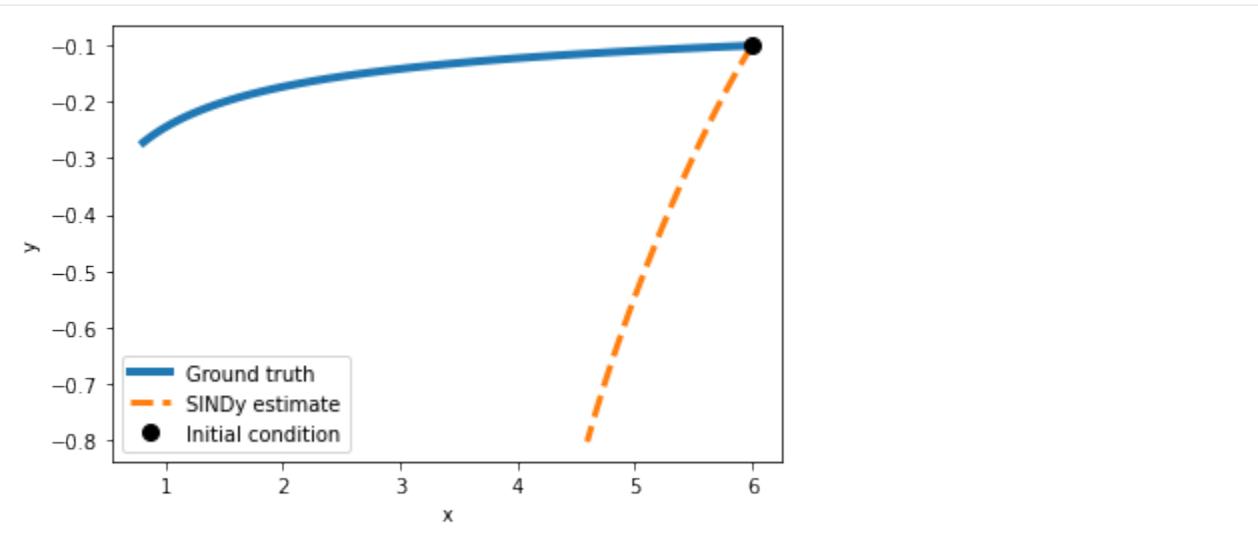
After fitting the model can be inspected with the *print* member function.

```
[19]: model.print()  
  
(x)' = 0.772 sin(1 x) + 2.097 cos(1 x) + -2.298 sin(1 y) + -3.115 cos(1 y)  
(y)' = 1.362 sin(1 y) + -0.222 cos(1 y)
```

Let us verify this on some test data:

```
[20]: import matplotlib.pyplot as plt  
def plot_simulation(model, x0, y0):  
    t_test = np.linspace(0, 1, 100)  
    x_test = x0 * np.exp(-2 * t_test)  
    y_test = y0 * np.exp(t_test)  
  
    sim = model.simulate([x0, y0], t=t_test)  
  
    plt.figure(figsize=(6, 4))  
    plt.plot(x_test, y_test, label="Ground truth", linewidth=4)  
    plt.plot(sim[:, 0], sim[:, 1], "--", label="SINDy estimate", linewidth=3)  
    plt.plot(x0, y0, "ko", label="Initial condition", markersize=8)  
    plt.xlabel("x")  
    plt.ylabel("y")  
    plt.legend()  
    plt.show()
```

```
[21]: x0 = 6  
y0 = -0.1  
plot_simulation(model, x0,y0)
```



As we may have expected, a Fourier basis is not optimal for this problem. Let's try a different basis function!

Exercise 1 Let's investigate what happens if we choose a different basis function.

Implement a SINDy algorithm class with a Fourier basis by initializing `ps.SINDy()` with the `feature_library` attribute set to `ps.PolynomialLibrary()`. Fit the instantiated algorithm with the `.fit(X, t=t)` method.

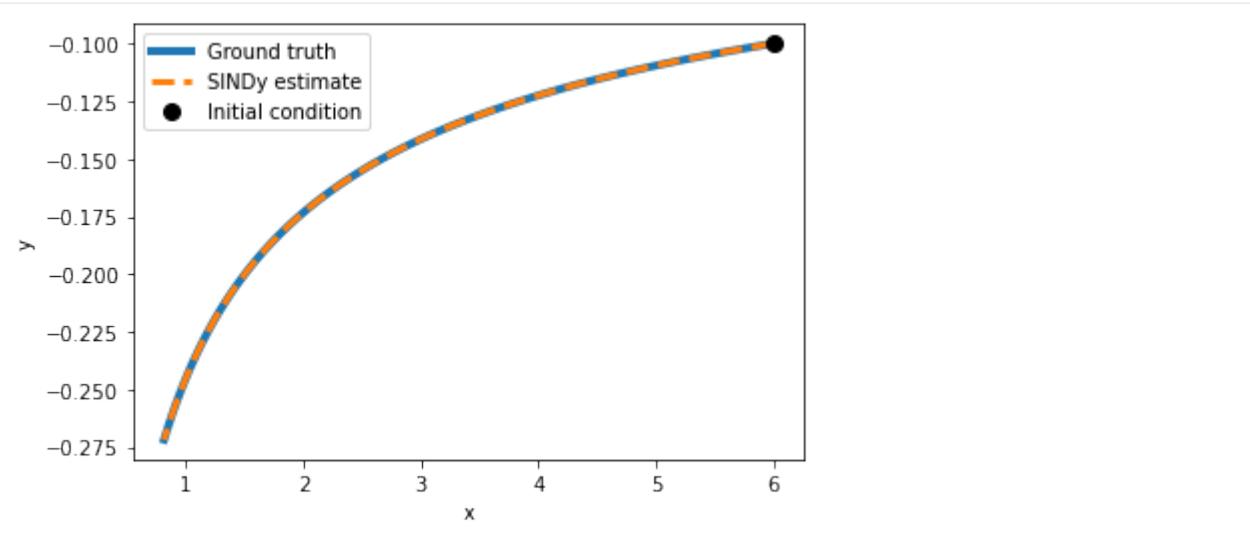
```
[103]: model_1 = ps.SINDy(
    differentiation_method=ps.FiniteDifference(order=2),
    feature_library=ps.PolynomialLibrary(degree=1),
    optimizer=ps.STLSQ(threshold=0.2),
    feature_names=['x', 'y'])
)
model_1.fit(X, t=t)

[103]: SINDy(differentiation_method=FiniteDifference(),
            feature_library=PolynomialLibrary(degree=1), feature_names=['x', 'y'],
            optimizer=STLSQ(threshold=0.2))
```

```
[104]: model_1.print()

(x)' = -2.000 x
(y)' = 1.000 y
```

```
[105]: x0 = 6
y0 = -0.1
plot_simulation(model_1, x0,y0)
```



As we can see in this (simple) case, the model has perfectly identified the underlying equation by using polynomial basis functions.

Our problem was a first-order polynomial differential equation, so it is only natural that regression with exactly these model assumptions succeeds.

Exercise 1.1 At which degree is SINDy not able anymore to correctly identify the equation? At which degree do the predictions on a test set diverge? Test this by successively increasing the degree of the polynomial.

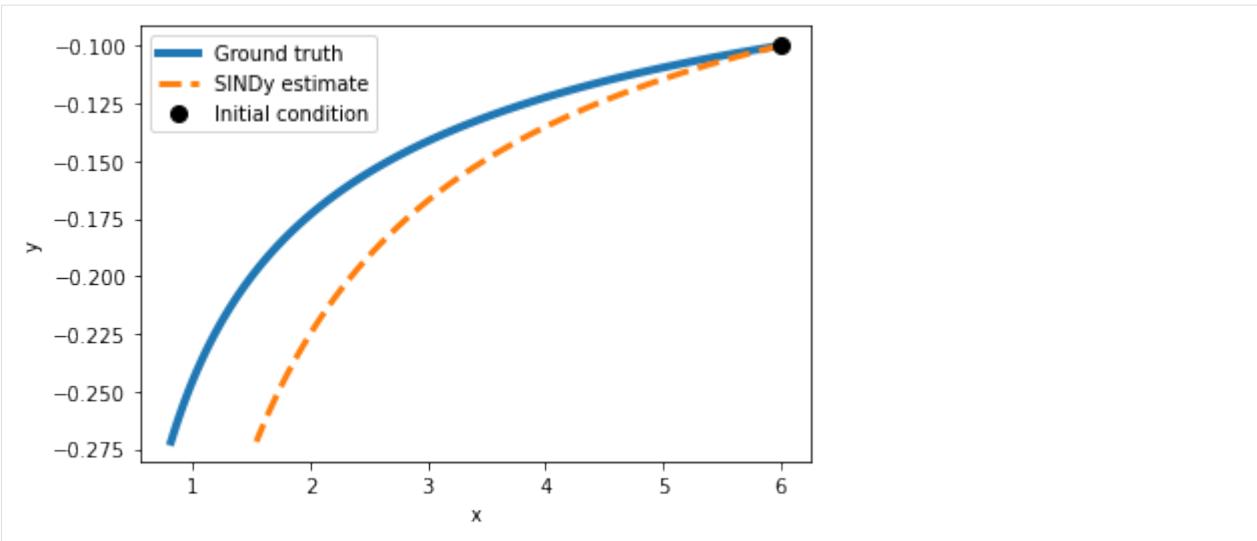
```
[106]: model_1 = ps.SINDy(
    differentiation_method=ps.FiniteDifference(order=2),
    feature_library=ps.PolynomialLibrary(degree=4),
    optimizer=ps.STLSQ(threshold=0.2),
    feature_names=["x", "y"])
)
model_1.fit(X, t=t)
```

```
[106]: SINDy(differentiation_method=FiniteDifference(),
    feature_library=PolynomialLibrary(degree=4), feature_names=['x', 'y'],
    optimizer=STLSQ(threshold=0.2))
```

```
[107]: model_1.print()

(x)' = -1.280 x + -0.960 x^2 y^2
(y)' = 1.000 y
```

```
[108]: x0 = 6
y0 = -0.1
plot_simulation(model_1, x0,y0)
```



Exercise 1.2 What happens if you set the threshold too low? What if it is too high?

STLSQ : Attempts to minimize the objective function $\|y - Xw\|_2^2 + \alpha\|w\|_2^2$ by iteratively performing least squares and masking out elements of the weight that are below a given threshold.

threshold : minimum magnitude for a coefficient in the weight vector. Coefficients with magnitude below the threshold are set to zero.

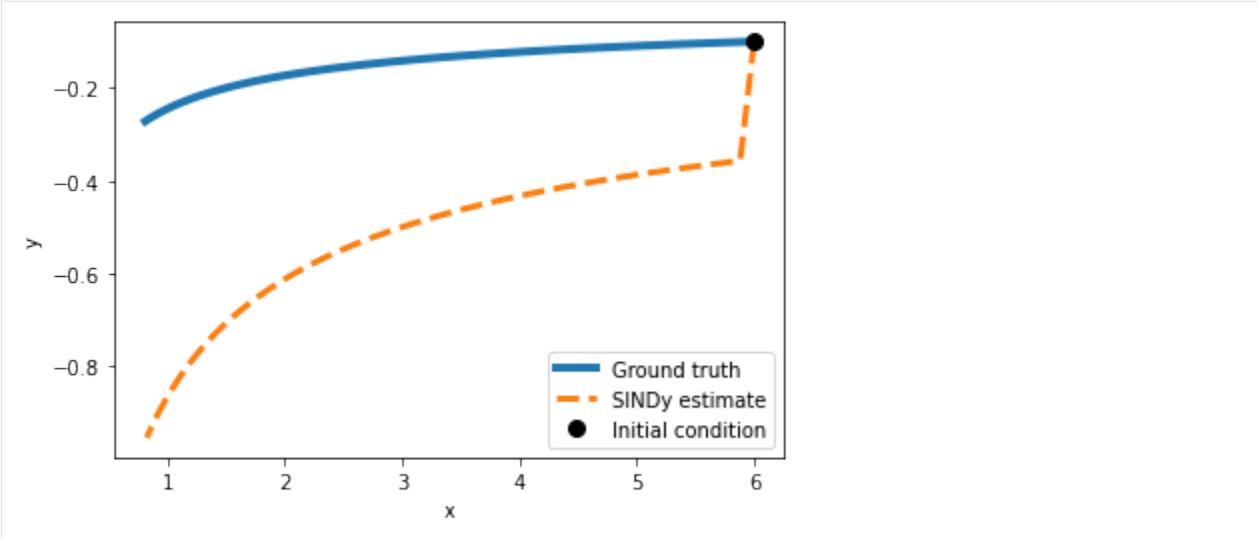
```
[109]: model_1 = ps.SINDy(
    differentiation_method=ps.FiniteDifference(order=2),
    feature_library=ps.PolynomialLibrary(degree=4),
    optimizer=ps.STLSQ(threshold=0.1),
    feature_names=['x', 'y'])
)
model_1.fit(X, t=t)

[109]: SINDy(differentiation_method=FiniteDifference(),
            feature_library=PolynomialLibrary(degree=4), feature_names=['x', 'y'],
            optimizer=STLSQ())
```

```
[110]: model_1.print()

(x)' = -1.277 x + 0.001 x^2 + -0.002 x y + -0.005 x^2 y + -0.958 x^2 y^2
(y)' = 3063674878.195 y + -4084899836.260 x y^3
```

```
[111]: x0 = 6
y0 = -0.1
plot_simulation(model_1, x0,y0)
```



4.39.5 Example: Lorenz attractor

For testing SINDy on the Lorenz attractor we first simulate a trajectory from the ground truth equations:

```
[112]: import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from mpl_toolkits.mplot3d import Axes3D

rho = 28.0
sigma = 10.0
beta = 8.0 / 3.0
dt = 0.01

def f(state, t):
    x, y, z = state
    return sigma * (y - x), x * (rho - z) - y, x * y - beta * z

state0 = [1.0, 1.0, 1.0]
time_steps = np.arange(0.0, 40.0, dt)

x_train = odeint(f, state0, time_steps)
```

```
[113]: model = ps.SINDy(
    optimizer=ps.STLSQ(threshold=0.05),
    feature_library=ps.PolynomialLibrary(degree=2),
)

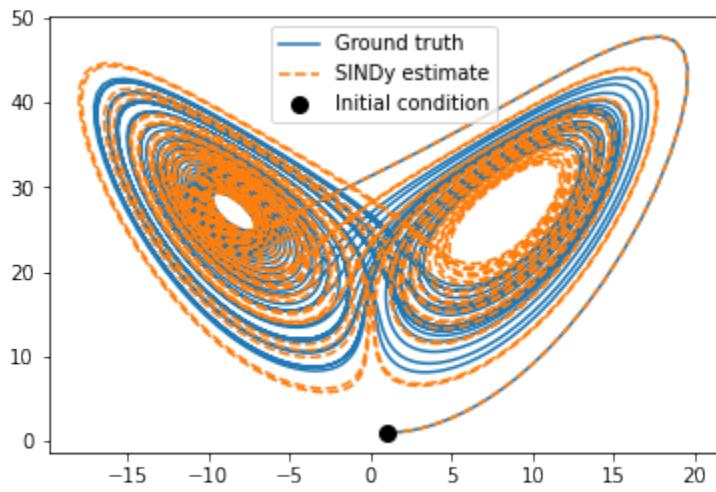
model.fit(x_train, t=dt)

x_sim = model.simulate(x_train[0], time_steps)
```

```
[114]: model.print()
```

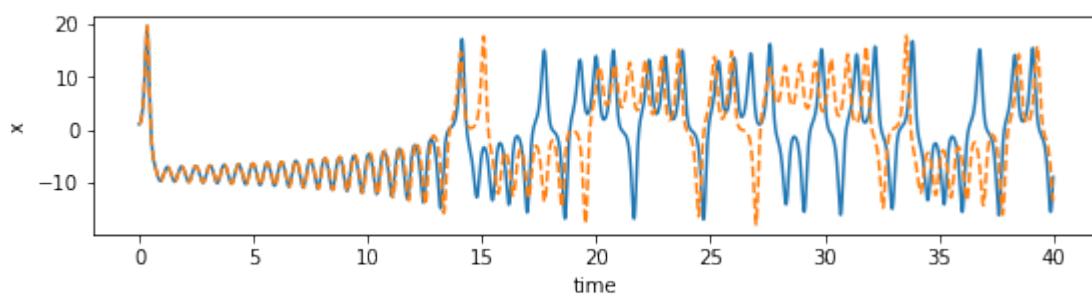
```
(x0)' = -9.977 x0 + 9.977 x1
(x1)' = 27.806 x0 + -0.962 x1 + -0.995 x0 x2
(x2)' = -2.659 x2 + 0.997 x0 x1
```

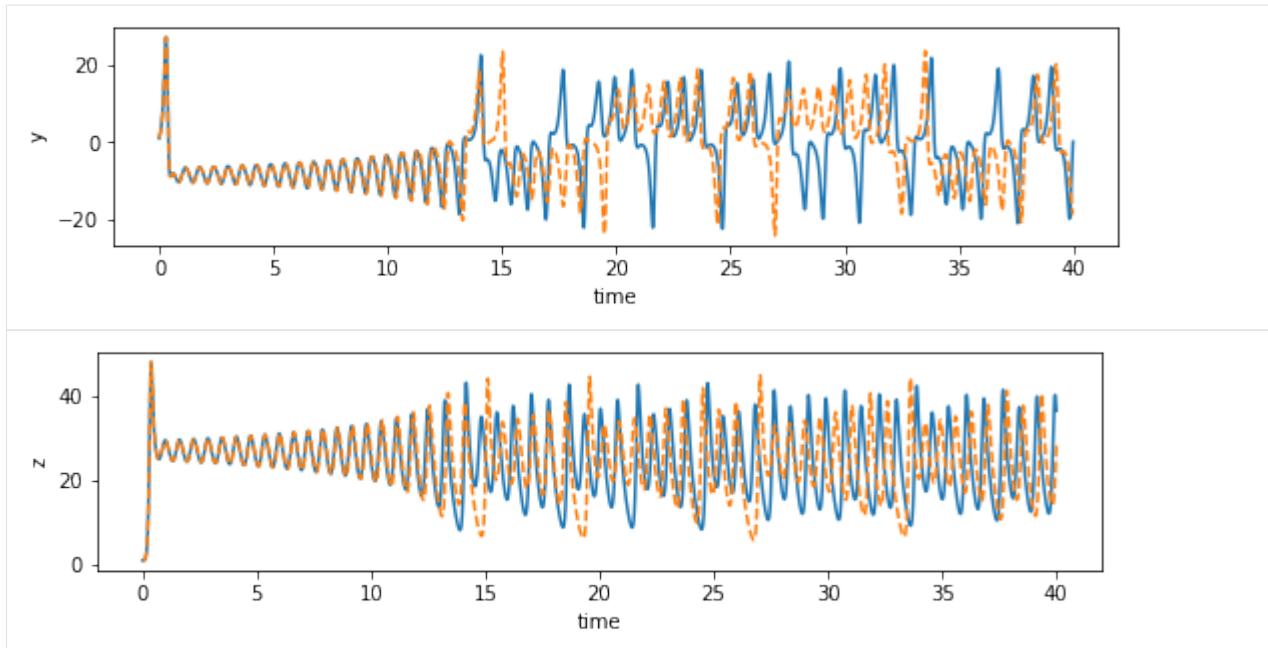
```
[115]: plt.figure(figsize=(6, 4))
plt.plot(x_train[:, 0], x_train[:, 2], label='Ground truth')
plt.plot(x_sim[:, 0], x_sim[:, 2], '--', label='SINDy estimate')
plt.plot(x_train[0, 0], x_train[0, 2], "ko", label="Initial condition", markersize=8)
plt.legend()
plt.draw()
plt.show()
```



```
[116]: def plot_dimension(dim, name):
    fig = plt.figure(figsize=(9, 2))
    ax = fig.gca()
    ax.plot(time_steps, x_train[:, dim])
    ax.plot(time_steps, x_sim[:, dim], "--")
    plt.xlabel("time")
    plt.ylabel(name)

plot_dimension(0, 'x')
plot_dimension(1, 'y')
plot_dimension(2, 'z')
```





We can see that the prediction tracks the ground truth data very faithfully, until it diverges (which is inevitable by the chaotic nature of the system).

4.39.6 Challenges

The previous examples have been very basic with a lot of implicit assumptions. Applying SINDy to real-world data is more challenging on multiple fronts. The challenges arise from: 1. **Data**. How much and what quality of data is needed? Per default, SINDy relies on clean data, rapidly sampled in time, with little noise. Needs to be clean enough to compute derivatives. But these requirements can be traded off against each other: e.g., if the data is clean, little is fine. Conversely, for ample but noisy data, one can average out the noise by integration. 2. **Coordinates**. Which variables to measure? Use expert domain knowledge if possible. Alternatively, apply singular value decomposition to find most relevant variables. Or train a deep autoencoder in combination with SINDy: >Champion, Kathleen, Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. 2019. “Data-Driven Discovery of Coordinates and Governing Equations.” *Proceedings of the National Academy of Sciences of the United States of America* 116 (45): 22445–51. <https://doi.org/10.1073/pnas.1906995116>.

3. **Library**. What library terms capture the dynamics best? Practically, start simple, e.g. with linear terms, then quadratic etc.. One may also use domain knowledge. Be careful, to not expand the library too big, to avoid linear dependence of columns resulting in an ill-conditioned matrix $\Theta(X)$. Prior knowledge on physics can yield symmetry constraints that cross out symmetry-incompatible terms in library.
4. **Optimization**. Which algorithm to use? How do I actually find sparse terms? Simplest is LASSO, in practice Sequential Threshold Least Squares (iteratively hard-thresholding coefficients) works better. Here, one may also incorporate *physics-informed* constraints: for example one can enforce energy conservation or global stability directly in the optimization.

In the past years, a lot has been achieved in this regard and SINDy has been successful in a variety of real-world applications, including discovering equations for plasma fluid dynamics that previously had not been analytically formulated due to the complexity of the problem.

Sources

- <https://pysindy.readthedocs.io/en/latest/#how-it-works>
- https://pysindy.readthedocs.io/en/latest/examples/2_introduction_to_sindy.html
- Part 1: <https://www.youtube.com/watch?v=NxAn0oglMVw>
- Part 2: <https://www.youtube.com/watch?v=8-hoWTJwmrE>
- Part 3: <https://www.youtube.com/watch?v=1vrsBg92Xzo>
- Part 4: https://www.youtube.com/watch?v=MmMNQe_EtCw
- Part 5: <https://www.youtube.com/watch?v=pY2iJnngk4g>

4.39.7 Reservoir Computing

Reservoir computing is a simple method for training recurrent neural networks without backpropagation through time and the associated notorious vanishing and exploding gradient problems. The basic steps are:

1. Randomly initialize recurrent neural network weights.
2. Fix hidden connection weights.
3. Train linear output layer with linear regression.

More formally, the state of the recurrent neural network of M neurons is given by its hidden activations $\mathbf{h} \in \mathbb{R}^M$, which are connected by randomly initialized and fixed matrix \mathbf{W}_h . An input sequence $\mathbf{X}_{1:\tau}$ is embedded by a linear map \mathbf{W}_i to the state \mathbf{h}_τ :

$$\begin{aligned}\mathbf{h}_1 &= \sigma(\mathbf{W}_h \mathbf{h}_0 + \mathbf{W}_i \mathbf{x}_1) \\ &\vdots \quad \vdots \\ \mathbf{h}_\tau &= \sigma(\mathbf{W}_h \mathbf{h}_{\tau-1} + \mathbf{W}_i \mathbf{x}_\tau).\end{aligned}$$

Then, the reservoir can linearly predict the next input $\mathbf{x}_{\tau+1}$, based on the hidden state \mathbf{h}_τ :

$$\hat{\mathbf{x}}_{\tau+1} = \mathbf{W}_o \mathbf{h}_\tau, \tag{4.99}$$

with \mathbf{W}_o mapping from hidden activation to output. Only the parameters of this output matrix \mathbf{W}_o are trained. Hence, the optimal values can be obtained analytically via ridge regression (with regularization parameter λ) as:

$$\mathbf{W}_o = (\mathbf{H}^T \mathbf{H} + \lambda \mathbf{I})^{-1} \mathbf{H}^T \mathbf{X},$$

where \mathbf{H} are the hidden states, \mathbf{I} the identity matrix, and \mathbf{X} the targets of the regression.

4.39.8 Example: Lorenz Attractor

```
[117]: from scipy import sparse

radius = 0.6
sparsity = 0.01
input_dim = 3
reservoir_size = 1000
n_steps_prerun = 10
regularization = 1e-2
sequence = x_train
```

Randomly initialize the network weights:

```
[118]: weights_hidden = sparse.random(reservoir_size, reservoir_size, density=sparsity)
eigenvalues, _ = sparse.linalg.eigs(weights_hidden)
weights_hidden = weights_hidden / np.max(np.abs(eigenvalues)) * radius

weights_input = np.zeros((reservoir_size, input_dim))
q = int(reservoir_size / input_dim)
for i in range(0, input_dim):
    weights_input[i * q:(i + 1) * q, i] = 2 * np.random.rand(q) - 1

weights_output = np.zeros((input_dim, reservoir_size))
```

Embed the sequence into the hidden state of the network:

```
[119]: def initialize_hidden(reservoir_size, n_steps_prerun, sequence):
    hidden = np.zeros((reservoir_size, 1))
    for t in range(n_steps_prerun):
        input = sequence[t].reshape(-1, 1)
        hidden = np.tanh(weights_hidden @ hidden + weights_input @ input)
    return hidden

def augment_hidden(hidden):
    h_aug = hidden.copy()
    h_aug[::2] = pow(h_aug[::2], 2.0)
    return h_aug

hidden = initialize_hidden(reservoir_size, n_steps_prerun, sequence)
hidden_states = []
targets = []

for t in range(n_steps_prerun, len(sequence) - 1):
    input = np.reshape(sequence[t], (-1, 1))
    target = np.reshape(sequence[t + 1], (-1, 1))
    hidden = np.tanh(weights_hidden @ hidden + weights_input @ input)
    hidden = augment_hidden(hidden)
    hidden_states.append(hidden)
    targets.append(target)

targets = np.squeeze(np.array(targets))
hidden_states = np.squeeze(np.array(hidden_states))
```

Ridge regression to obtain the linear output layer weights:

```
[120]: weights_output = (np.linalg.inv(hidden_states.T@hidden_states + regularization * np.
    eye(reservoir_size)) @ hidden_states.T@targets).T
```

```
[121]: def predict(sequence, n_steps_predict):
    hidden = initialize_hidden(reservoir_size, n_steps_prerun, sequence)
    input = sequence[n_steps_prerun].reshape((-1, 1))
    outputs = []

    for t in range(n_steps_prerun, n_steps_prerun + n_steps_predict):
        hidden = np.tanh(weights_hidden @ hidden + weights_input @ input)
```

(continues on next page)

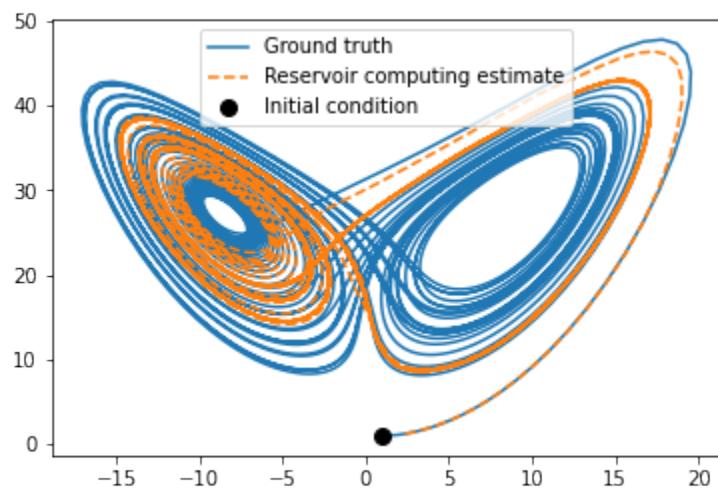
(continued from previous page)

```
hidden = augment_hidden(hidden)
output = weights_output @ hidden
 input = output
outputs.append(output)
return np.array(outputs)
```

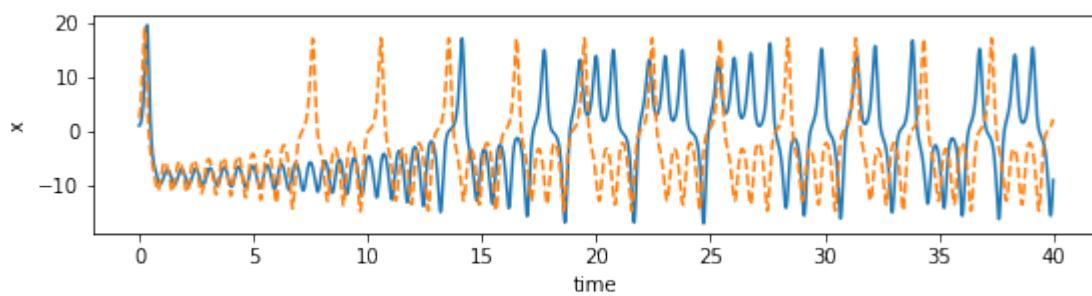
[122]: `x_sim = predict(sequence, 4000)`

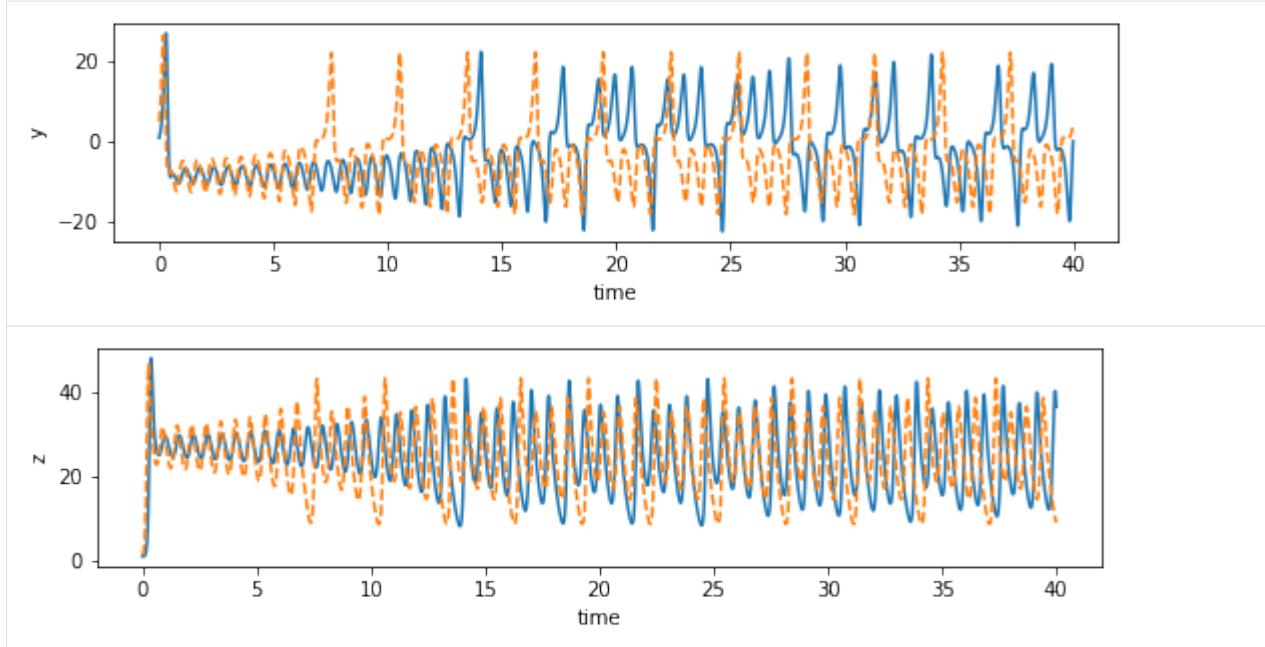
[123]: `plt.figure(figsize=(6, 4))
plt.plot(x_train[:4000, 0], x_train[:4000, 2], label="Ground truth")
plt.plot(x_sim[:, 0], x_sim[:, 2], '--', label="Reservoir computing estimate")
plt.plot(x_train[0, 0], x_train[0, 2], "ko", label="Initial condition", markersize=8)

plt.legend()
plt.show()`



[124]: `plot_dimension(0, 'x')
plot_dimension(1, 'y')
plot_dimension(2, 'z')`





A successful application of reservoir computing to data-driven estimation of nonlinear dynamical systems is for example:

Jaideep Pathak et al., “Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach,” Physical Review Letters 120, no. 2 (January 12, 2018): 024102, <https://doi.org/10.1103/PhysRevLett.120.024102>.

4.40 DNN - Tutorial 2 Part II: Physics inspired Machine Learning

Partial Differential Equation (PDE) Solvers

Filled notebook:

Authors: Ilze Amanda Auzina

Before we start one practical thing:

Later in this tutorial we will use datasets to train our own Neural PDE Solver. It takes a little bit of time for the datasets to be uploaded, so let's do that first! (So by the time you get to the coding part the data is already in your environment)

- Go to: [drive](#)
- Download the 3 datasets on your local machine (click on ‘Download anyway’ when prompted second time)
- Upload them here: click on ‘Files’ → right click ‘Upload’ (click ‘ok’ on the prompt)

4.40.1 PDE vs ODE: What are PDEs?

A **partial derivative** of a function (with several variables) is a derivative with respect to one of its variables while the other variables are held constant.

For example take the Heat equation (1 dimension):

$$\frac{\delta T}{\delta t}(x, t) = \frac{\delta^2 T}{\delta x^2}(x, t)$$

Here we have one partial derivative w.r.t time (left hand side, LHS), and another partial derivative w.r.t to space (right hand side, RHS), as our function $T(x,t)$ has 2 variables. On the LHS variable x is held constant, while on the RHS variable t is held constant.

In particular, the above PDE describes how the temperature $T(x,t)$ changes in time for a 1 dimensional object, such as a metal rod. If we look at the PDE equation above it says that the *rate of change of temperature in time* is the same as the *the second order derivate of the spatial domain*.

This makes sense as the second order derivate gives you a measure of how a value compares to the average of its neighbours. For example, if the neighbors of a given point x_i are very cold compared to the point itself then the rate of change in temperature in time for this point will be *faster*.

In the image below we have visualized the partial derivatives of the 1d Heat equation. We can investigate the rate of change of the temperature in two domains: spatial (see Fig 1) and time (see Fig 2). (credits: 3Blue1Brwon series source)

```
[2]: import matplotlib.pyplot as plt
import matplotlib.image as mpimg

img_space = mpimg.imread('images/SpaceDerivative.png')
img_time= mpimg.imread('images/TimeDerivative.png')
f, axarr = plt.subplots(1,2, figsize=(30,30))
axarr[0].set_axis_off()
axarr[0].set_title('Fig 1: Space Derivative', fontsize=25)
axarr[1].set_axis_off()
axarr[1].set_title('Fig 2: Time Derivative', fontsize=25)
axarr[0].imshow(img_space)
axarr[1].imshow(img_time);
```

Fig 1: Space Derivative

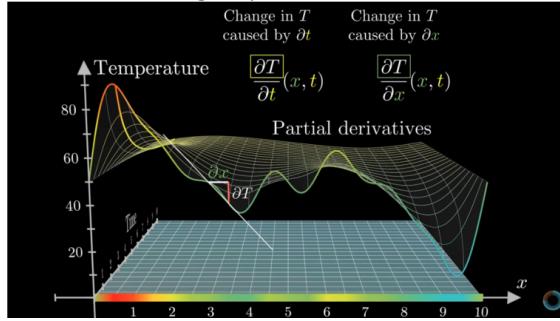
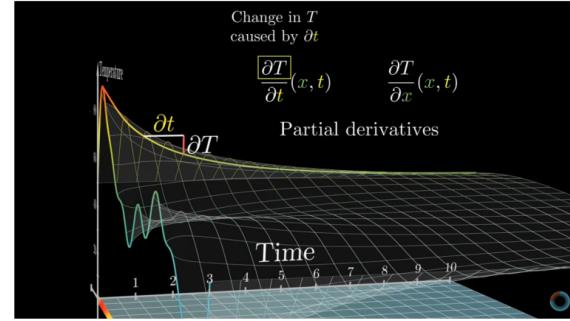


Fig 2: Time Derivative



As you can see above each of the derivatives tells only a part of the story on how the function changes, therefore we call them partial derivatives. In principle, PDEs could be seen as a system of ordinary differential equations.

In **dynamical systems** compared to traditional ML we are most interested in how our function $T(x,t)$ changes over time.

The above 1D Heat equation can be also extended to more spatial directions by extending the right hand side:

$$\frac{\delta T}{\delta t}(x, y, z, t) = \alpha * \left(\frac{\delta^2 T}{\delta x^2}(x, y, z, t) + \frac{\delta^2 T}{\delta y^2}(x, y, z, t) + \frac{\delta^2 T}{\delta z^2}(x, y, z, t) \right)$$

Why do we care about PDEs?

They allow us to model (predict) real life systems that influence our life: - economics: [Black-Scholes equation](#) that governs price evolution - climate: [Weather prediction](#), think about how your google weather app knows what's the weather going to be like tomorrow - in general, the entire physical world around us

To conclude:

PDEs tell us how a function with several variables changes over time. Most of the physical world is governed by PDEs.

4.40.2 How to solve PDEs?

Let's now shift our focus on a specific PDE, for example, the **Korteweg-de Vries (KdV)** equation.

The KdV equation (1885) is a model of shallow water waves:

In the above x again represents a spatial location, $U(x)$ is the value of x given a function U , and we can see the evolution of the system in time.

The PDE equation governing the above simulation is:

$$\frac{\delta U}{\delta t}(x, t) = -U(x, t) * \frac{\delta U}{\delta x}(x, t) - \frac{\delta^3 U}{\delta x^3}(x, t)$$

Similar to the heat equation in the introduction our function U has two dimensions, x and t , where x is the spatial domain and t is the time domain. The $\frac{\delta^3 U}{\delta x^3}(x, t)$ is a [dispersive term](#) and $U(x, t) * \frac{\delta U}{\delta x}(x, t)$ is a [non-linear convection term](#).

When we are talking about solving a PDE what we want to do is to find a function $U(x, t)$ that would: - satisfy the above given PDE - meet the boundary conditions - meet the initial conditions

These additional constraints, boundary and initial conditions, restrict the function $U(x, t)$ that is valid for the given PDE equation

Boundary Conditions

The boundary conditions express the behaviour of a function on the boundary (border) of its area of definitions. For example, you constrain that at both ends of the spatial domain x_0 and x_L (where L is the end of your spatial domain) the function remains constant for all time:

$$\frac{\delta U}{\delta x}(x_0, t) = \frac{\delta U}{\delta x}(x_L, t) = 0$$

$\forall t$

Initial Conditions

An initial condition is similar to boundary conditions, but now in the time direction. For example, now you constrain what value your function $U(x, t)$ must return at timepoint $t=0$. For the KdV simulation above the initial condition was:

$$U(x, 0) = \cos(\pi x) \quad \forall x$$

Solving PDE

Similarly to ODEs the traditional way of solving a given PDE equation is via [numerical solvers](#).

In the below cells we will implement a simple numerical solver in order to solve the above KdV equation

Numerical Solvers

```
[3]: #Import relevant packages
%matplotlib inline
import matplotlib
from matplotlib.pyplot import cm
import numpy as np
from typing import Optional
from scipy.integrate import solve_ivp
from scipy.fftpack import diff as psdiff

from matplotlib import animation
import seaborn as sns
from IPython.display import HTML
```

Generate Initial Conditions

Initial conditions are sampled from a distribution over truncated Fourier series with random coefficients $\{A_k, \ell_k, \phi_k\}_k$ as

$$U(x, 0) = \sum_{k=1}^N A_k \sin(2\pi\ell_k x/L + \phi_k).$$

```
[4]: def generate_params() -> (int, np.ndarray, np.ndarray, np.ndarray):
    """
    Returns parameters for initial conditions.

    Args:
        None
    Returns:
        int: number of Fourier series terms
        np.ndarray: amplitude of different sine waves
        np.ndarray: phase shift of different sine waves
        np.ndarray: frequency of different sine waves
    """
    N = 10 #Number of different waves
    lmin, lmax = 1, 3 #sine frequencies for intial conditions
    A = (np.random.rand(1, N) - 0.5)
    phi = 2.0*np.pi*np.random.rand(1, N)
    l = np.random.randint(lmin, lmax, (1, N))
    return (N, A, phi, l)

def initial_conditions(x: np.ndarray, L: int, params: Optional[list]=None) -> np.ndarray:
    """
    Return initial conditions based on initial parameters.

    Args:
        x (np.ndarray): input array of spatial grid
        L (float): length of the spatial domain
        params (Optinal[list]): input parameters for generating initial conditions
    Returns:
        np.ndarray: initial condition
```

(continues on next page)

(continued from previous page)

```
"""
if params is None:
    params = generate_params()
N, A, phi, l = params
u = np.sum(A * np.sin((2 * np.pi * l * x[:, None] / L) + phi), -1)
return u
```

Solve via Method of Lines (MOL)

In MOL all but the temporal dimension are discretized. Having the spatial derivatives numerically implemented results in a set of coupled ODEs for the time domain, which can be solved by using integration schemes of ODE solving.

Concretely, for getting numerical spatial derivatives, we use [pseudospectral methods](#), where the derivatives are computed in the frequency domain by first applying a fast fourier transform (FFT) to the data, then multiplying by the appropriate values and converting back to the spatial domain with the inverse FFT.

Mathematically this works since the Fourier transform of the n th derivative is given by:

$$\widehat{f^{(n)}}(\xi) = \mathcal{F}\left(\frac{d^n}{dx^n} f(x)\right) = (2\pi i \xi)^n \widehat{f}(\xi),$$

where $\widehat{f^{(n)}}$ is the mathematical denotation of fourier transform of a function f , the transform variable ξ represents frequency, $f(x)$ is an absolutely continuous differentiable function, and both f and its derivative f' are integrable (for more information see [wikipedia](#) section Functional relations). This method of differentiation in the Fourier space is implemented by the `diff` function in the module `scipy.fftpack`. For integration in time we use an **implicit Runge-Kutta method** of Radau IIA family, order 5.

```
[5]: # Spatial Derivatives
def kdv_pseudospectral(t: float, u: np.ndarray, L: float) -> np.ndarray:
    """
    Compute spatial derivatives for the KdV equation, using a pseudospectral method,
    descretization in x.

    Args:
        t (float): time point
        u (np.ndarray): 1D input field
        L (float): length of the spatial domain
    Returns:
        np.ndarray: reconstructed pseudospectral time derivative
    """

    # Compute the x derivatives using the pseudo-spectral method.
    ux = psdiff(u, order=1, period=L)
    uxxx = psdiff(u, order=3, period=L)
    # Compute du/dt.
    dudt = -u*ux - uxxx
    return dudt
```

Solve for KdV trajectory

```
[6]: # Set the size of the domain, and create the discretized grid.
np.random.seed(1)
L = 128
N = 2**7
x = np.linspace(0, (1-1.0/N)*L, N)

# Set the tolerance of the solver
tol = 1e-6

# Set the initial conditions.
u0 = initial_conditions(x, L)

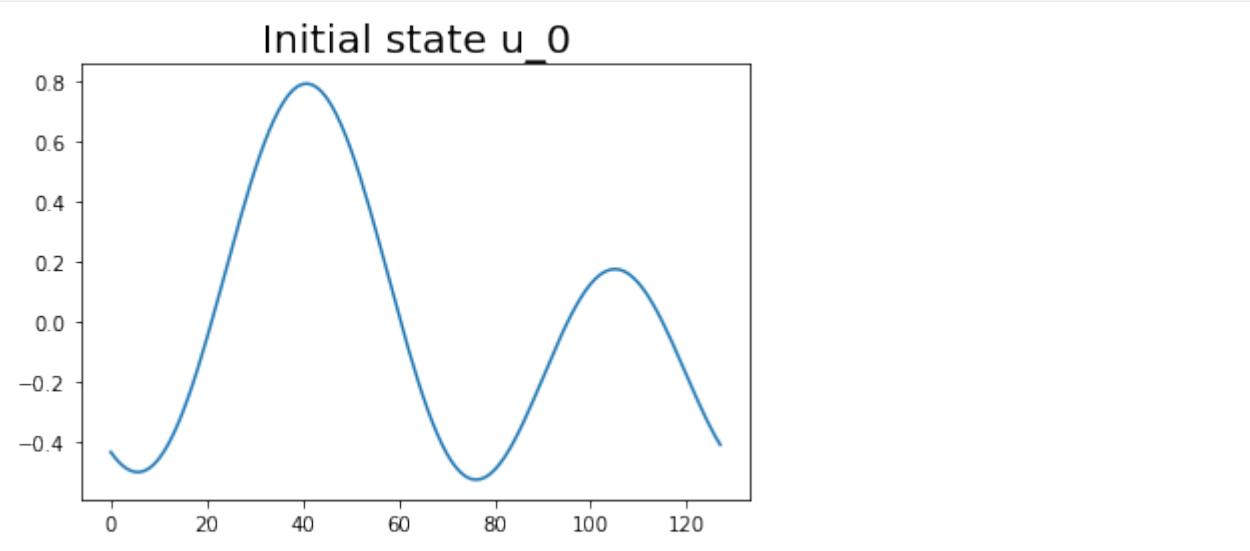
# Set the time sample grid.
T = 100.
t = np.linspace(0, T, 200)

# Compute the solution using kdv_pseudospectral as spatial solver
sol_ps = solve_ivp(fun=kdv_pseudospectral,
                    t_span=[t[0], t[-1]],
                    y0=u0,
                    method='Radau',
                    t_eval=t,
                    args=(L,),
                    atol=tol,
                    rtol=tol)
```

Visualize the solved KdV

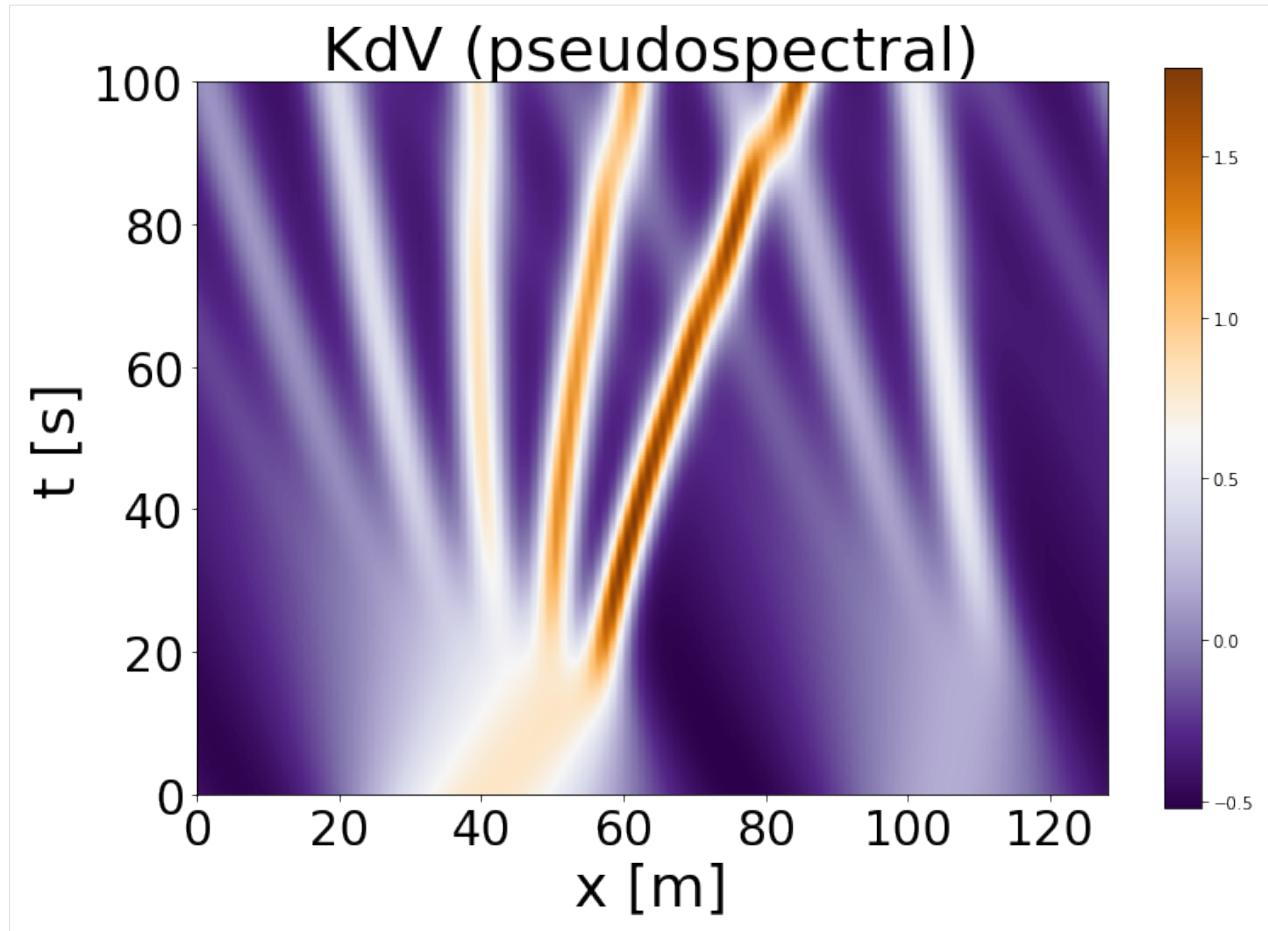
Let's first plot our initial state:

```
[7]: plt.plot(u0)
plt.title('Initial state u_0', fontsize=20)
plt.show();
```



Now let's plot the entire time evolution, where time is as the y axis, spatial domain as the x axis, and the function value $U(x,t)$ is the colorbar

```
[8]: # Let's look at the trajectory obtained by using the pseudospectral spatial solver
if sol_ps.success:
    t_ps = sol_ps.y.T[::-1]
    plt.figure(figsize=(12,8))
    plt.imshow(t_ps, extent=[0,L,0,T], cmap='PuOr_r')
    plt.colorbar()
    plt.title('KdV (pseudospectral)', fontsize=36)
    plt.xlabel('x [m]', fontsize=34)
    plt.ylabel('t [s]', fontsize=34)
    plt.yticks(fontsize=28)
    plt.xticks(fontsize=28)
    plt.show()
```



In the figure above we can see the time evolution of our initial state $U(x, t_0)$. As we only have one spatial domain, each x point is a coordinate in our spatial domain, for example, the bottom left point would be the value of $U(x_0, t_0)$. The color indicates the value of our function $U(\cdot)$, and the more brown the color the *higher* the wave. Interpret the *stripes* you see as waves and as time progresses the waves get sharper.

Feel free to play around with the settings under `generate_params()` and see how the simulation changes!

Let's create our own animation!

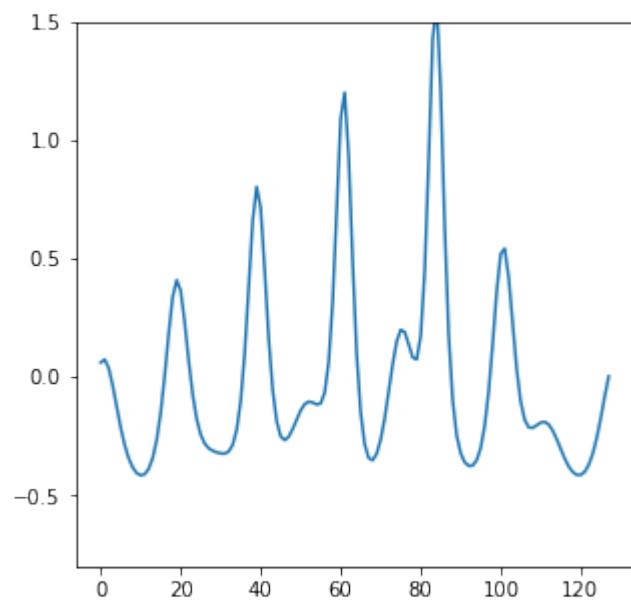
```
[9]: plt.rcParams["animation.html"] = "jshtml"
```

```
[10]: fig, ax = plt.subplots(figsize=(5,5))
t_ps_anim = np.flipud(t_ps)

def animate(t):
    plt.cla()
    plt.ylim([-0.8, 1.5])
    plt.plot(t_ps_anim[t,:])

matplotlib.animation.FuncAnimation(fig, animate, frames=200, interval=20)
```

```
[10]: <matplotlib.animation.FuncAnimation at 0x167e35ee0>
```



In the animation, on the y-axis we have the output of the function, $U(x, t)$. As the animation plays we can see the waves of our initial state to become more peaked.

Why do we care about Numerical Solvers?

This story is two-fold: - We need numerical solvers to generate data to train our ML models - Numerical solvers have higher accuracy than NN, hence, they provide a baseline for expected accuracy for a given Neural PDE Solver

And in general the field of numerical solvers is about 100 years old, definitely there are things that we can learn from them!

4.40.3 Neural Networks as PDE Solvers

The initial interest of why we would want to replace traditional numerical solvers with deep learning is because traditional PDE solvers, such as finite element methods and finite difference methods, rely on discretizing the space into a very fine mesh. As a result traditional solvers have the following drawbacks: - The error scales steeply with the resolution. We need a high resolution to get good approximations. - The computation and storage steeply scale with the resolution (i.e. the size of the grid). - When the equation is solved on one discretization, we cannot change the discretization anymore.

So the question is: *can we use neural networks to improve upon the drawbacks of numerical solvers?*

In short, the answer is **yes** and in the upcoming section we will explore on how to construct such a network.

What is our network's aim?

What we would like our deep neural network to do is that given our initial and/or boundary conditions to directly output the solution, similar to image-to-image mapping. See below, we have an initial state *Initial Vorticity* and we want to predict the '*image*' at t_{15} .

```
[11]: img_pde = mpimg.imread('images/PDEimg-img.png')
caption = "Credits: <a href='https://zongyi-li.github.io/blog/2020/fourier-pde/">https://zongyi-li.github.io/blog/2020/fourier-pde/"
```

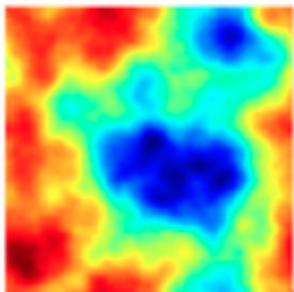
(continues on next page)

(continued from previous page)

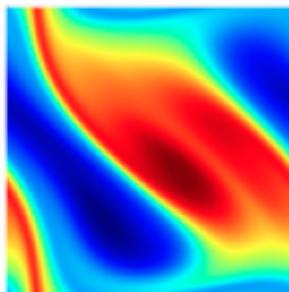
```
plt.imshow(img_pde)
plt.text(0, 0, caption, ha='left', fontsize=10)
plt.axis('off');
```

Credits: <https://zongyi-li.github.io/blog/2020/fourier-pde/>

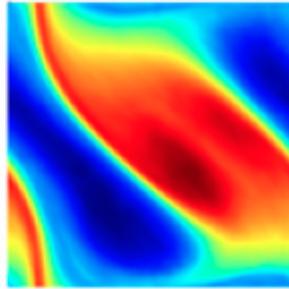
Initial Vorticity



t=15



Prediction



However, when we talk about PDEs we consider our input and output space to be **functions**. What we mean by this is that we have some initial function U that takes as input x, y and t , $U(x, y, t_0)$ respectively. In the above image that would be the *Initial Vorticity* state. We want our model to output perhaps the resulting function 15 time steps later $A(x, y, t_{15})$, so the state to the right. As such, **images could be considered as functions**. In the context of standard ML an image is a function of light defined on a continuous region, instead of 32x32 pixel vectors.

Hence, we want our neural network, G (parameterised by θ) to map between function spaces:

$$G_\theta : UA$$

In order to do so let me introduce to you the concept of **neural operators**.

Neural Operators

Let's break down this term “**Neural Operators (NO)**” in 2 parts: (i) neural and (ii) operator. With the first one you probably already are familiar, so let's start with the second one.

Operator

The term **operator** comes from mathematics:

an operator is generally a mapping or function that acts on elements of a space to produce elements of another space

Indeed, that is exactly what he have defined as the goal of our NN.

So what is the benefit of rephrasing the img-to-img problem to func-to-func? **Generalizability**.

The neural operator is **mesh-independent**, this is different from standard ML methods such as CNNs. The func-to-func mapping perspective allow us to: (i) first use a numerical solver to generate some low-resolution data, (ii) then train a neural operator on this data, (iii) but now the learned NO is able to give also high-resolution predictions, despite being trained on low-resolution data. This makes both, the training and the evaluation, easier.

In short: the benefit of operator persepctive is that now we are parameterizing the model in function space - **it learns the continuous functions instead of discretized vectors**

Neural

The neural part is that just like NN: our neural operator will also consist of linear transformations, K , and non-linear activations, σ .

A standard deep neural network can be written as:

$$y = (K_{ll}\sigma_1 K_0)x$$

where defines function composition, y is the output vector, x is the input vector, K are linear or convolutional layers, and σ are the activation functions such as ReLU.

As we dicussed for operators our inputs and outputs are functions, hence we need to adjust the above to neural operator framework:

$$a(x) = (K_{ll}\sigma_1 K_0)u(x)$$

now our output vector y is replaced by a function a , and our input vector x is replaced by a function u . Both functions, u and a are able to take inputs at different discretizations (28x28, or 256x256), hence allowing the neural operator to be discretization independent.

In order for the above to be true we need to adjust the formulation of our linear layers, K , as it has to be able to take functions as input. We reformulate K as a kernel integral operator, changing the layer architecture to:

$$\begin{aligned} u_{t+1}(x) &= \sigma(Wu_t(x) + (Ku_t)(x)) \\ (Ku_t)(x) &:= k(x, y)u(y)dy \end{aligned}$$

where k is a kernel function and W is the bias term. The only question remaining is how to define the kernel itself, $k(x, y)$. Recent research has investigated multiple alternatives, however the kernel that has brought the most promising results is the **Fourier Neural Operator**.

Fourier Neural Operator

What Li et al. (2020) proposed is to set the above kernel integral operator, $(Ku_t)(x)$, as a convolution operator defined in Fourier space. Why is this desirable? - it's fast: convolution in one domain (spatial domain) equals **point-wise multiplication** in the frequency domain (**convolution theorem**) - it's efficient: the inputs and outputs of PDEs are continuous functions, so it's more efficient to represent them in Fourier space.

Hence, the **Fourier Integral Operator** is:

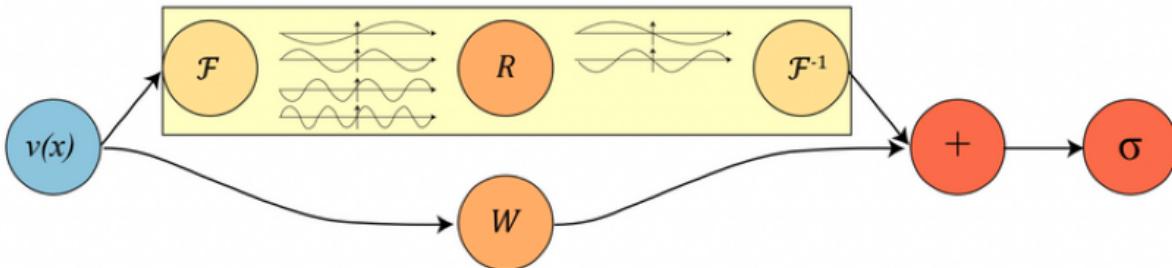
$$(Ku_t)(x) = {}^{-1} (R_\phi(u_t))(x)$$

where R_ϕ is the linear transform of the lower Fourier modes. Let's break down what is happening in the above function: - first we do a Fourier transform: $(Fu_t)(x)$ - then a linear transform: $R \cdot (\cdot)$ - and lastly we perform an inverse fourier transform: $F^{-1}(\cdot)$

In the image below we have depicted a complete **Fourier layer** ($v(x)$ is $u(x)$ in our notation):

```
[12]: img_fourier = mpimg.imread('images/fourierlayer.png')
caption = "Credits: https://zongyi-li.github.io/blog/2020/fourier-pde/"
plt.figure(figsize=(15, 10))
plt.imshow(img_fourier)
plt.text(0, 0, caption, ha='left', fontsize=10)
plt.axis('off');
```

Credits: <https://zongyi-li.github.io/blog/2020/fourier-pde/>



As you can see in the above figure R_ϕ acts as a linear transformation on the lower frequency modes and sets the higher modes to zeros. Nonetheless, despite the fact that Fourier Integral Operator on its own loses the higher frequency modes and works only with periodic boundary conditions, the operator as a whole does not suffer from this: > The bias term helps to recover the non-periodic boundaries, and the non-linear activations help to recover the higher frequency modes. The activations act on the spatial domain, which is also the reason why we need the inverse fourier transform

Resolution Invariance

The Fourier layers are discretization-invariant, because they can learn from and evaluate functions which are discretized in an arbitrary way. Since parameters are learned directly in Fourier space, resolving the functions in physical space simply amounts to projecting on the basis of wave functions which are well-defined everywhere on the space. This allows us to transfer among discretization.

4.40.4 Time to code a Neural PDE Solver

We will train our Neural PDE Solver on the KdV PDE that was introduced earlier in this notebook.

The Data

the data is located in your runtime environment. If you click on ‘Files’ on the left side, you will see the downloaded files: - KdV_train_512_easy.h5 - KdV_valid_easy.h5 - KdV_test_easy.h5

Each dataset contains 512 trajectories. The length, L, of the spatial domain is 128, while the spatial resolution of the dataset is 256. We stimulate each trajectory for 100 time steps, with a temporal resolution of 140.

KdV equation initial conditions are generated by the same principle as in the earlier example in the notebook

Fourier Integral Operator

Let’s first implement the euqation of the **Fourier Integral Operator** that we discussed in the previous section.

```
[13]: import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.parameter import Parameter
```

```
[14]: #####
# 1d Fourier Integral Operator
#####
class SpectralConv1d(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, modes: int):
        super(SpectralConv1d, self).__init__()
        """
        Initializes the 1D Fourier layer. It does FFT, linear transform, and Inverse FFT.
        Args:
            in_channels (int): input channels to the FNO layer
            out_channels (int): output channels of the FNO layer
            modes (int): number of Fourier modes to multiply, at most floor(N/2) + 1
        """
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.modes = modes
        self.scale = (1 / (in_channels*out_channels))
        self.weights = nn.Parameter(self.scale * torch.rand(in_channels, out_channels, self.modes, dtype=torch.cfloat))

    # Complex multiplication
    def compl_mul1d(self, input, weights):
        """
        Complex multiplication of the Fourier modes.
        [batch, in_channels, x], [in_channel, out_channels, x] -> [batch, out_channels, x]
        """
        Args:
            input (torch.Tensor): input tensor of size [batch, in_channels, x]
            weights (torch.Tensor): weight tensor of size [in_channels, out_channels, self.modes, x]
```

(continues on next page)

(continued from previous page)

```

    Returns:
        torch.Tensor: output tensor with shape [batch, out_channels, x]
    """
    return torch.einsum("bix,iox->box", input, weights)

def forward(self, x: torch.Tensor) -> torch.Tensor:
    """
    Fourier transformation, multiplication of relevant Fourier modes,
    ↵ backtransformation
    Args:
        x (torch.Tensor): input to forward pass os shape [batch, in_channels, x]
    Returns:
        torch.Tensor: output of size [batch, out_channels, x]
    """
    batchsize = x.shape[0]
    # Fourier transformation
    x_ft = torch.fft.rfft(x)

    # Multiply relevant Fourier modes
    out_ft = torch.zeros(batchsize, self.out_channels, x.size(-1)//2 + 1, device=x.
    ↵ device, dtype=torch.cfloat)
    out_ft[:, :, :self.modes] = self.compl_mul1d(x_ft[:, :, :self.modes], self.
    ↵ weights)

    #Return to physical space
    x = torch.fft.irfft(out_ft, n=x.size(-1))
    return x

```

Now once we have our integral kernel, we are ready to build our neural network.

Overall Network Architecture

We will follow the original architecture as described by the [FNO paper](#): the network is constructed by stacking four Fourier layers with ReLU activations.

In addition, we will add some minor improvements as described by [LPSDA paper](#): we will predict in ‘batches’ (multiple time points)

```
[41]: #####
# 1d Fourier Network
#####
class FNO1d(nn.Module):
    def __init__(self, modes, width, time_future, time_history):
        super(FNO1d, self).__init__()

        """
        The overall network. It contains 4 layers of the Fourier layer.
        1. Lift the input to the desire channel dimension by self.fc0 .
        2. 4 layers of the integral operators u' = (W + K)(u).
           W defined by self.w; K defined by self.conv .
        3. Project from the channel space to the output space by self.fc1 and self.fc2 .

        input: a driving function observed at T timesteps + 1 locations (u(1, x), ...,
        ↵ u(T, x), x).

```

(continues on next page)

(continued from previous page)

```

input shape: (batchsize, x=s, c=2)
output: the solution of a later timestep
output shape: (batchsize, x=s, c=1)
"""

self.modes = modes
self.width = width
self.time_future = time_future
self.time_history = time_history
self.fc0 = nn.Linear(self.time_history+1, self.width)

self.conv0 = SpectralConv1d(self.width, self.width, self.modes)
self.conv1 = SpectralConv1d(self.width, self.width, self.modes)
self.conv2 = SpectralConv1d(self.width, self.width, self.modes)
self.conv3 = SpectralConv1d(self.width, self.width, self.modes)
self.w0 = nn.Conv1d(self.width, self.width, 1)
self.w1 = nn.Conv1d(self.width, self.width, 1)
self.w2 = nn.Conv1d(self.width, self.width, 1)
self.w3 = nn.Conv1d(self.width, self.width, 1)

self.fc1 = nn.Linear(self.width, 128)
self.fc2 = nn.Linear(128, self.time_future)

def forward(self, u):
    grid = self.get_grid(u.shape, u.device)
    x = torch.cat((u, grid), dim=-1)
    x = self.fc0(x)
    x = x.permute(0, 2, 1)

    x1 = self.conv0(x)
    x2 = self.w0(x)
    x = x1 + x2
    x = F.gelu(x)

    x1 = self.conv1(x)
    x2 = self.w1(x)
    x = x1 + x2
    x = F.gelu(x)

    x1 = self.conv2(x)
    x2 = self.w2(x)
    x = x1 + x2
    x = F.gelu(x)

    x1 = self.conv3(x)
    x2 = self.w3(x)
    x = x1 + x2

    x = x.permute(0, 2, 1)
    x = self.fc1(x)
    x = F.gelu(x)
    x = self.fc2(x)
    return x

```

(continues on next page)

(continued from previous page)

```
def get_grid(self, shape, device):
    batchsize, size_x = shape[0], shape[1]
    gridx = torch.tensor(np.linspace(0, 1, size_x), dtype=torch.float)
    gridx = gridx.reshape(1, size_x, 1).repeat([batchsize, 1, 1])
    return gridx.to(device)
```

Helper functions

```
[42]: import h5py
import torch
from typing import Tuple
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch import nn

#function to torch dataloader from the dataset
def create_dataloader(data_string: str, mode: str, nt: int, nx: int, batch_size:int, num_
→workers:int):
    try:
        dataset = HDF5Dataset(data_string, mode, nt=nt, nx=nx)
        loader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_workers=num_
→workers)
    except:
        raise Exception("Datasets could not be loaded properly")

    return loader

#Function to format the data in the correct format
def to_coords(x: torch.Tensor, t: torch.Tensor) -> torch.Tensor:
    """
    Transforms the coordinates to a tensor X of shape [time, space, 2].
    Args:
        x: spatial coordinates
        t: temporal coordinates
    Returns:
        torch.Tensor: X[..., 0] is the space coordinate (in 2D)
                      X[..., 1] is the time coordinate (in 2D)
    """
    x_, t_ = torch.meshgrid(x, t)
    x_, t_ = x_.T, t_.T
    return torch.stack((x_, t_), -1)

#Helper class to open the .h5 formated file
class HDF5Dataset(Dataset):
    """
    Load samples of an PDE Dataset, get items according to PDE.
    """

    def __init__(self, path: str,
                 mode: str,
                 nt: int,
```

(continues on next page)

(continued from previous page)

```

        nx: int,
        dtype=torch.float64,
        load_all: bool=False):
    """Initialize the dataset object.
Args:
    path: path to dataset
    mode: [train, valid, test]
    nt: temporal resolution
    nx: spatial resolution
    shift: [fourier, linear]
    pde: PDE at hand
    dtype: floating precision of data
    load_all: load all the data into memory
"""
super().__init__()
f = h5py.File(path, 'r')
self.mode = mode
self.dtype = dtype
self.data = f[self.mode]
self.dataset = f'pde_{nt}-{nx}'

if load_all:
    data = {self.dataset: self.data[self.dataset][:]}
    f.close()
    self.data = data

def __len__(self):
    return self.data[self.dataset].shape[0]

def __getitem__(self, idx: int) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    """
    Returns data items for batched training/validation/testing.
    Args:
        idx: data index
    Returns:
        torch.Tensor: data trajectory used for training/validation/testing
        torch.Tensor: dx
        torch.Tensor: dt
    """
    u = self.data[self.dataset][idx]
    x = self.data['x'][idx]
    t = self.data['t'][idx]
    dx = self.data['dx'][idx]
    dt = self.data['dt'][idx]

    if self.mode == "train":
        X = to_coords(torch.tensor(x), torch.tensor(t))
        sol = (torch.tensor(u), X)

    u = sol[0]

```

(continues on next page)

(continued from previous page)

```

X = sol[1]
dx = X[0, 1, 0] - X[0, 0, 0]
dt = X[1, 0, 1] - X[0, 0, 1]
else:
    u = torch.from_numpy(u)
    dx = torch.tensor([dx])
    dt = torch.tensor([dt])
return u.float(), dx.float(), dt.float()

#function to create x - y data pairs: 20 past timepoints as x, 20 future timepoints as y
def create_data(datapoints: torch.Tensor, start_time: list, time_future: int, time_
history: int) -> Tuple[torch.Tensor, torch.Tensor]:
"""
    Getting data of PDEs for training, validation and testing.
    Args:
        datapoints (torch.Tensor): trajectory input
        start_time (int list): list of different starting times for different_
    ↪ trajectories in one batch
        pf_steps (int): push forward steps
    Returns:
        torch.Tensor: neural network input data
        torch.Tensor: neural network labels
    """
    data = torch.Tensor()
    labels = torch.Tensor()
    # Loop over batch and different starting points
    # For every starting point, we take the number of time_history points as training_
    ↪ data
    # and the number of time future data as labels
    for (dp, start) in zip(datapoints, start_time):
        end_time = start+time_history
        d = dp[start:end_time]
        target_start_time = end_time
        target_end_time = target_start_time + time_future
        l = dp[target_start_time:target_end_time]

        data = torch.cat((data, d[None, :]), 0)
        labels = torch.cat((labels, l[None, :]), 0)

    return data, labels

```

Train the Neural PDE Solver

First we will define the configurations of our experiment and create the data loader for training, validation and test data

```
[43]: #####
# configurations
#####
#Experiment Settings
ntrain = 512 #train samples
```

(continues on next page)

(continued from previous page)

```

nt = 140 #temporal resolution
nx = 256 #spatial resoltuion
time_history = 20 #time steps to be considered as input to the solver
time_future = 20 #time steps to be considered as output of the solver
device = 'cpu' #change to cpu if no cuda available
print_interval = 20

#model parameters
modes = 16 # number of Fourier modes to multiply
width = 64 # input and output channels to the FNO layer
batch_size = 16
num_epochs = 1 #set to one so faster computation, in principle 20 is best
learning_rate = 0.0001
lr_decay = 0.4
num_workers = 0

```

[35]:

```

#####
# read data
#####
train_loader = create_dataloader("KdV_train_10_1_3_512_easy.h5", mode="train", nt=nt, ↴
    ↴ nx=nx, batch_size=batch_size, num_workers=num_workers)
valid_loader = create_dataloader("KdV_valid_10_1_3_easy.h5", mode="valid", nt=nt, nx=nx, ↴
    ↴ batch_size=batch_size, num_workers=num_workers)
test_loader = create_dataloader("KdV_test_10_1_3_easy.h5", mode = "test", nt=nt, nx=nx, ↴
    ↴ batch_size=batch_size, num_workers=num_workers)

```

Initialize the model

[44]:

```

import operator
from functools import reduce
# print the number of parameters
def count_params(model):
    c = 0
    for p in list(model.parameters()):
        c += reduce(operator.mul,
                    list(p.size()+(2,) if p.is_complex() else p.size()))
    return c

# model
model = FNO1d(modes, width, time_future, time_history).to(device)
print('Number of paramters: {}'.format(count_params(model)))

```

Number of paramters: 553236

Select an optimizer

we also implement a scheduler ‘MultiStepLR’ that decays the learning rate, to learn more read ‘info <<https://hasty.ai/content-hub/mp-wiki/scheduler/multisteplr>>’

[45]:

```

# Optimizer
from torch import optim
optimizer = optim.AdamW(model.parameters(), lr=learning_rate)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[0, 5, 10, 15], ↴
    ↴ gamma=lr_decay)

```

(continues on next page)

(continued from previous page)

Select a training objective

```
[46]: criterion = torch.nn.MSELoss(reduction="none")
```

Below are functions for the training and test loop

```
[47]: import random
```

```
def bootstrap(x: torch.Tensor, Nboot: int=64, binsize: int=1) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    Bootstrapping mean or median to obtain standard deviation.
    Args:
        x (torch.Tensor): input tensor, which contains all the results on the different trajectories of the set at hand
        Nboot (int): number of bootstrapping steps, 64 is quite common default value
        binsize (int):
    Returns:
        torch.Tensor: bootstrapped mean/median of input
        torch.Tensor: bootstrapped variance of input
    """
    boots = []
    x = x.reshape(-1, binsize, *x.shape[1:])
    for i in range(Nboot):
        boots.append(torch.mean(x[torch.randint(len(x), (len(x),))], axis=(0, 1)))
    return torch.tensor(boots).mean(), torch.tensor(boots).std()

def training_loop(loader, model, optimizer, criterion, device, batch_size):
    """
    Training loop.
    Loop is over the mini-batches and for every batch we pick a random timestep.
    This is done iteratively for the number of timesteps in one training sample.
    One average, we therefore start at every starting point in every trajectory during one episode.
    Args:
        model (torch.nn.Module): neural network model
        optimizer (torch.optim): chosen optimizer
        loader (DataLoader): train/valid/test dataloader
        criterion (torch.nn.modules.loss): loss criterion
        device: device (cpu/gpu)
    """
    losses = []
    max_start_time = (nt - time_history) - time_future
    for u, _, _ in loader:
        optimizer.zero_grad()
        start_time = random.choices([t for t in range(time_history, max_start_time + 1, time_history)], k=batch_size)

        data, labels = create_data(u, start_time, time_history, time_future)
        data, labels = data.to(device), labels.to(device)
```

(continues on next page)

(continued from previous page)

```

# Change [batch, time, space] -> [batch, space, time]
data = data.permute(0, 2, 1)

#forward pass
pred = model(data)

loss = criterion(pred.permute(0, 2, 1), labels)
loss = loss.sum()
loss.backward()
losses.append(loss.detach() / batch_size)
optimizer.step()

losses = torch.stack(losses)
return losses

def test(loader, model, criterion, device, batch_size):
    """
    Test routine.
    Both step wise losses and enrolled forward losses are computed.
    Args:
        model (torch.nn.Module): neural network model
        loader (DataLoader): train/valid/test dataloader
        criterion (torch.nn.modules.loss): loss criterion
        device: device (cpu/gpu)
    Returns:
        (torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor): mean and normalized
        mean errors
        of full trajectory unrolling
    """
    losses, nlosses = [], []
    max_start_time = (nt - time_history) - time_future
    for u, _, _ in loader:
        losses_tmp, nlosses_tmp = [], []
        with torch.no_grad():
            # the first time steps are used for data augmentation according to time_
            # translate
            # we ignore these timesteps in the testing
            for start in range(time_history, max_start_time+1, time_future):

                end_time = start + time_history
                target_end_time = end_time + time_future
                if start == time_history:
                    data = u[:, start:end_time].to(device)
                    data = data.permute(0, 2, 1)
                else:
                    data = torch.cat([data, pred], -1)
                    data = data[..., -time_history:]

                labels = u[:, end_time: target_end_time].to(device)
                pred = model(data)

```

(continues on next page)

(continued from previous page)

```

loss = criterion(pred.permute(0, 2, 1), labels)
nlabels = torch.mean(labels ** 2, dim=-1, keepdim=True)
nloss = loss / nlabels
loss, nloss = loss.sum(), nloss.sum()
loss, nloss = loss / nx / batch_size, nloss / nx / batch_size
losses_tmp.append(loss)
nlosses_tmp.append(nloss)

losses.append(torch.sum(torch.stack(losses_tmp)))
nlosses.append(torch.sum(torch.stack(nlosses_tmp)))

losses = torch.stack(losses)
nlosses = torch.stack(nlosses)

mean, std = bootstrap(losses, 64, 1)
nmean, nstd = bootstrap(nlosses, 64, 1)
print(f'Unrolled forward losses: {mean:.4f} +- {std:.4f}')
print(f'Unrolled forward losses (normalized): {nmean:.4f} +- {nstd:.4f}')

return mean, std, nmean, nstd

```

Train the PDE solver

```
[48]: from datetime import datetime
min_val_loss = 10e30
test_loss, ntest_loss = 10e30, 10e30
test_loss_std, ntest_loss_std = 0., 0.

# Log file and save path for model
dateTimeObj = datetime.now()
timestring = f'{dateTimeObj.date().month}{dateTimeObj.date().day}{dateTimeObj.time()'.
    hour}{dateTimeObj.time().minute}'

save_path = f'FN01d_KdV_future{time_future}_time{timestring}.pt'

for epoch in range(num_epochs):
    print(f"Epoch {epoch}")
    ######
    # train model
    #####
    model.train()
    #run every epoch (twice) as often as we have number of time steps in one trajectory
    for i in range(nt*2):
        losses = training_loop(train_loader, model, optimizer, criterion, device, batch_
        size)
        if(i % print_interval == 0):
            print(f'Training Loss (progress: {i / (nt * 2):.2f}): {torch.mean(losses)}')

    ######
    # evaluate model on validation data

```

(continues on next page)

(continued from previous page)

```
#####
print("Evaluation on validation dataset:")
model.eval()
# Test the losses (full trajectory)
val_loss, _, _, _ = test(valid_loader,model,criterion, device, batch_size)

#####
# evaluate model on test
#####
if(val_loss < min_val_loss):
    print("Evaluation on test dataset:")
    test_loss, test_loss_std, ntest_loss, ntest_loss_std = test(test_loader,model,
criterion, device, batch_size)
    # Save model
    torch.save(model.state_dict(), save_path)
    print(f"Saved model at {save_path}\n")
    min_val_loss = val_loss

scheduler.step()
```

```
Epoch 0
Training Loss (progress: 0.00): 2225.366943359375
Training Loss (progress: 0.07): 317.81427001953125
Training Loss (progress: 0.14): 203.3621826171875
Training Loss (progress: 0.21): 130.92327880859375
Training Loss (progress: 0.29): 93.80006408691406
Training Loss (progress: 0.36): 68.2781982421875
Training Loss (progress: 0.43): 54.970794677734375
Training Loss (progress: 0.50): 47.671485900878906
Training Loss (progress: 0.57): 40.30413055419922
Training Loss (progress: 0.64): 31.2979793548584
Training Loss (progress: 0.71): 30.720670700073242
Training Loss (progress: 0.79): 27.46445083618164
Training Loss (progress: 0.86): 22.985952377319336
Training Loss (progress: 0.93): 20.814678192138672
Evaluation on validation dataset:
Unrolled forward losses: 4.6350 +- 0.5282
Unrolled forward losses (normalized): 5.8777 +- 0.4042
Evaluation on test dataset:
Unrolled forward losses: 3.5452 +- 0.3387
Unrolled forward losses (normalized): 5.2532 +- 0.2734
Saved model at FN0ld_KdV_future20_time4241851.pt
```

Predict Future time Step

Let's see how well our model performs compared to the underlying ground truth!

at the moment we are loading a model trained on 20 epochs, pulled from the github repo, if you want you can change the file path to the model you just trained in your notebook

```
[59]: # load model
modelpath = 'models/FN01d_KdV_samples512_future20_time424184.pt' #if you want, replace ↵
↪with the model you just have trained!
model = FN01d(modes=16, width=64, time_future=20, time_history=20).to(device)
model.load_state_dict(torch.load(modelpath, map_location = torch.device(device))) #if you ↵
↪want replace with the model you trained!
model.eval()

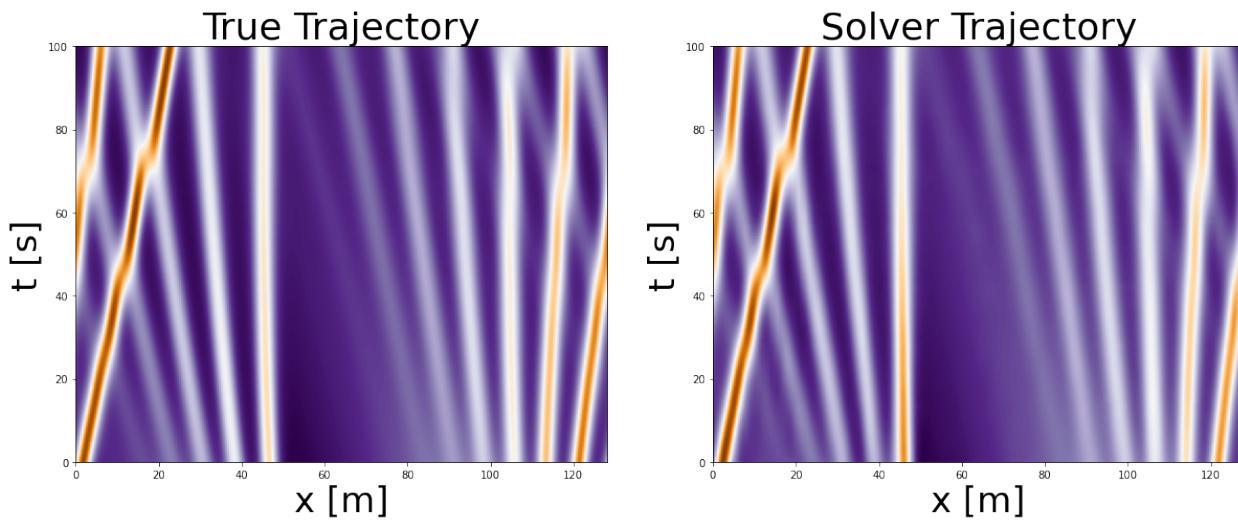
[59]: FN01d(
    (fc0): Linear(in_features=21, out_features=64, bias=True)
    (conv0): SpectralConv1d()
    (conv1): SpectralConv1d()
    (conv2): SpectralConv1d()
    (conv3): SpectralConv1d()
    (w0): Conv1d(64, 64, kernel_size=(1,), stride=(1,))
    (w1): Conv1d(64, 64, kernel_size=(1,), stride=(1,))
    (w2): Conv1d(64, 64, kernel_size=(1,), stride=(1,))
    (w3): Conv1d(64, 64, kernel_size=(1,), stride=(1,))
    (fc1): Linear(in_features=64, out_features=128, bias=True)
    (fc2): Linear(in_features=128, out_features=20, bias=True)
)
```

We will pass to our solver only the first 20 timesteps, after that the solver will predict autoregressively

```
[60]: start = 0
end_time = 20
trajectory = np.zeros(shape=(140, 256))

#sample from test and generate
with torch.no_grad():
    u, _, _ = next(iter(test_loader))
    true_trajectory = u[0]
    data = u[:1, start:end_time].to(device)
    trajectory[start:end_time,:] = torch.squeeze(data).cpu().detach().numpy()
    start += end_time
    end_time += time_future
    data = data.permute(0,2,1)
    while end_time <= 140:
        pred = model(data)
        data = pred
        trajectory[start:end_time,:] = torch.squeeze(data.permute(0,2,1)).cpu().detach().numpy()
        start = end_time
        end_time += time_future
```

```
[61]: #Plot True trajectory: produced by a numerical solver
# and Solver trajectory: produced by our NN
L=128
T=100
f, axarr = plt.subplots(1,2, figsize=(20,20))
axarr[0].set_title('True Trajectory', fontsize=36)
axarr[1].set_title('Solver Trajectory', fontsize=36)
axarr[0].imshow(true_trajectory, extent=[0,L,0,T], cmap='PuOr_r')
axarr[1].imshow(trajecotry, extent=[0,L,0,T], cmap='PuOr_r')
axarr[0].set_xlabel('x [m]', fontsize=34)
axarr[0].set_ylabel('t [s]', fontsize=34)
axarr[1].set_xlabel('x [m]', fontsize=34)
axarr[1].set_ylabel('t [s]', fontsize=34);
```



Execute the above 2 cells multiple times: the trajectory will change as we are using next function on the test dataset. You can see how well the model performs on different initial conditions (different wave formations).

As you can see after visual inspection our trained NN performs quite well on multiple initial conditions of the given PDE. Often in the literature this is highlighted as the benefit of Neural PDE Solvers → improved generalizability across multiple initial conditions. However, I would take this with a bit of a grain of salt as in order to train this network we first actually had to use a numerical solver that generated all the trajectories for these different initial conditions.

Zero-Shot Resolution

We trained our network on 256 resolution of the spatial domain:

```
[62]: u, _, _ = next(iter(train_loader))
print('Spatial resolution of training data: {}'.format(u[0].shape[1]))
print('Time resolution of training data: {}'.format(u[0].shape[0]))
```

Spatial resolution of training data: 256
Time resolution of training data: 140

Now let's see what happens if we input to our network a higher resolution, let's say **512**. We are going to keep the temporal resolution unchanged, 140

First let's generate such initial data and true trajectory by a numerical solver

We changed the parameter $N \rightarrow$ it defines the resolution of our 128 length rod

```
[76]: # Set the size of the domain, and create the discretized grid.
np.random.seed(3)
L = 128
N = 2**9
x = np.linspace(0, (1-1.0/N)*L, N)

# Set the tolerance of the solver
tol = 1e-6

# Set the initial conditions.
u0 = initial_conditions(x, L)

# Set the time sample grid.
T = 100.
t = np.linspace(0, T, 140)

# Compute the solution using kdv_pseudospectral as spatial solver
sol_ps_higher = solve_ivp(fun=kdv_pseudospectral,
                           t_span=[t[0], t[-1]],
                           y0=u0,
                           method='Radau',
                           t_eval=t,
                           args=(L,),
                           atol=tol,
                           rtol=tol)

true_trajectory_h = sol_ps_higher.y.T[::-1]
print('The spatial resolution {} and the temporal {}'.format(true_trajectory_h.shape[1],
                                                               true_trajectory_h.shape[0]))

The spatial resolution 512 and the temporal 140
```

Now let's use the first 20 timepoints as input to our network, and allow the network to predict the latter.

```
[89]: start = 0
end_time = 20
trajectory_h = np.zeros(shape=(140, 512))
correct_dir = np.flipud(true_trajectory_h)
trajectory_h[start:end_time, :] = correct_dir[start:end_time, :]

#take initial datapoints and generate the whole trajectory
with torch.no_grad():
    while start < 120:
        data = torch.from_numpy(correct_dir[start:end_time, :]).to(device)
        data = torch.unsqueeze(data, 0)
        data = data.permute(0, 2, 1)
        pred = model(data.float())
        start += 20
        end_time += 20
        trajectory_h[start:end_time, :] = torch.squeeze(pred.permute(0, 2, 1)).cpu().detach() .
        numpy()
```

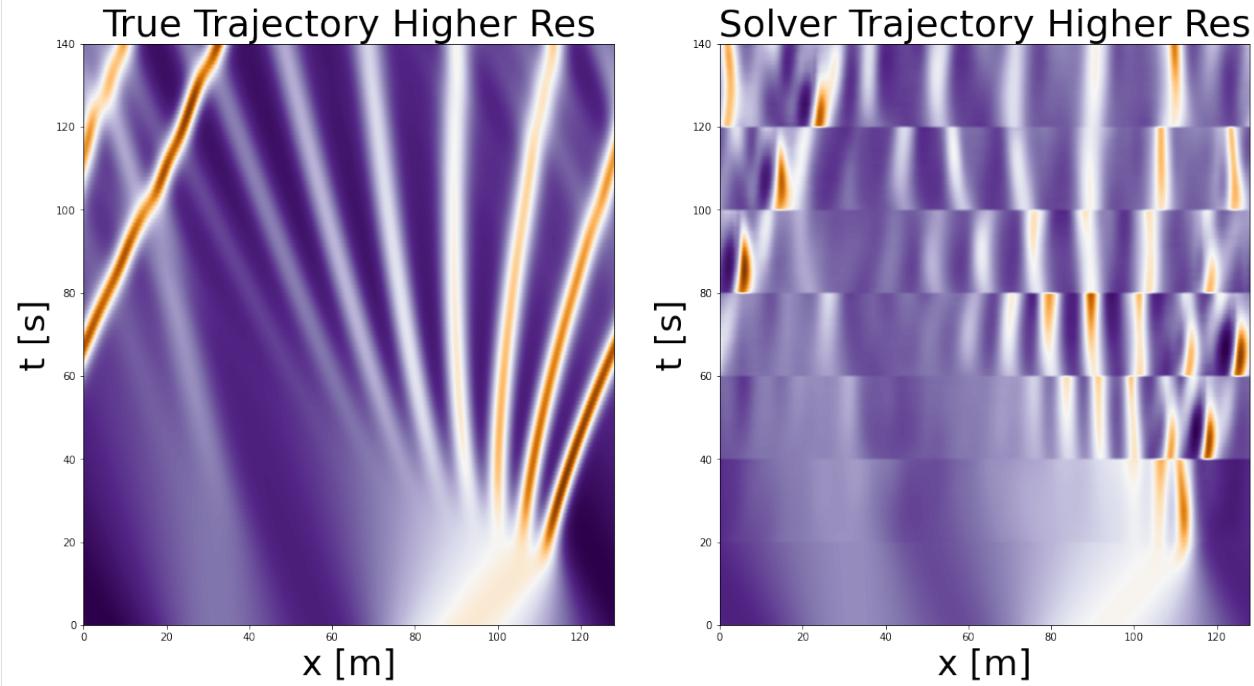
(continues on next page)

(continued from previous page)

```
trajectory_h = np.flipud(trajectory_h)
```

[90]: #Let's plot the true and predicted trajectory

```
L=128
T=140
f, axarr = plt.subplots(1,2, figsize=(20,20))
axarr[0].set_title('True Trajectory Higher Res', fontsize=36)
axarr[1].set_title('Solver Trajectory Higher Res', fontsize=36)
axarr[0].imshow(true_trajectory_h, extent=[0,L,0,T], cmap='PuOr_r')
axarr[1].imshow(trajectory_h, extent=[0,L,0,T], cmap='PuOr_r')
axarr[0].set_xlabel('x [m]', fontsize=34)
axarr[0].set_ylabel('t [s]', fontsize=34)
axarr[1].set_xlabel('x [m]', fontsize=34)
axarr[1].set_ylabel('t [s]', fontsize=34);
```



As we can see our PDE Solver is able to take as input higher resolution and also output higher resolution. As such, it has the quality of being **resolution-invariant**.

In the same time we see that the output of our predictions isn't too accurate. One way how we can improve this is by adding two additional channels as input, the spatial and temporal resolutions (dx and dt). For further details see the [LPSDA paper](#)

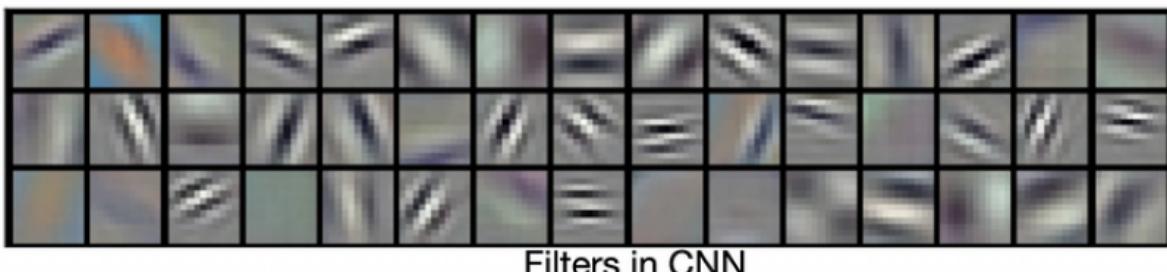
4.40.5 Fourier Filters compared to CNNs

One of the reasons proposed by Li et al. why fourier filters are more suited for dynamical problems than CNNs is because filters in convolution neural networks are usually local, while fourier filters are global sinusoidal functions. As such, CNNs are better in capturing local patterns such as edges and shapes, while fourier filters are better for representing continuous functions (see image below).

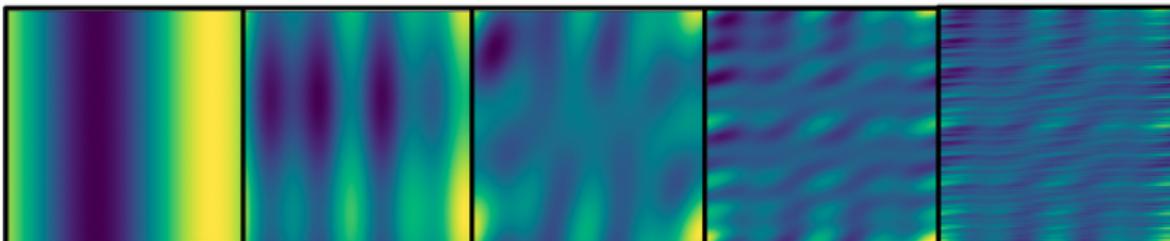
```
[92]: img_fourier = mpimg.imread('images/filters.png')
caption = "Credits: 

```

Credits: <https://zongyi-li.github.io/blog/2020/fourier-pde/>



Filters in CNN



Fourier Filters

4.40.6 Related Work

Now, after reading all of this you might be wondering: *how did they think of fourier filter in the first place?*

Truth be told, the authors actually first tried multiple different neural operators: [graph NO](#), [multipole graph NO](#), [low-rank NO](#), and from all of these the fourier NO turned out to be the most promising. However, it is quite interesting to think why so. Actually in math Fourier transform is often used to solve differential equations as it can reformulate them as problems that are easier to solve. Hence, intuitively it makes sense that also for neural networks → operating on a problem in fourier space might be easier.

The paper, based on which this tutorial was created, was published in ICLR 2020. Since then more researchers have built upon this topic. Here I would like to highlight some of them:

- [Message Passing](#): uses a different underlying architecture but has nice tricks for longer/more stable rollouts
- [LEADS](#): Model generalization across environments
- [Factorized FNO](#): this paper implements well known tricks from deep learning in order to create deeper FNO architectures

- Lie Point Symmetry: this paper uses pre-existing knowledge of PDE symmetries to improve generalizability and rollouts

4.40.7 Investigate Yourself

I hope that after this tutorial you have gotten more curious about modelling dynamical systems, as well as we have broaden your perspective on deep learning possibilities!

Below are a few things that you can try to change in the code to play around with it:

- Simply play around with the parameters: number of modes to keep, hidden layer size, the depth of your network etc. How does that affect your output?
- Implement a CNN, how does it perform in comparison?
- What happens for very long rollouts?
- How does the model trained on KdV perform on a different PDE? Is it able to generalize?
- We were working on a 1d problem, can you adjust it to 2d?

4.40.8 Acknowledgements

This tutorial was inspired/based upon on the following great resources:

- Z.Li Blogpost
- J.Brandsetter github

4.41 DS - Dynamical Systems & Neural ODEs

Filled notebook:

Empty notebook:

Authors: Riccardo Valperga (Part 1) & Miltos Kofinas (Part 2)

4.41.1 0. Introduction

This is a tutorial on dynamical systems, Ordinary Differential Equations (ODEs) and numerical solvers, and Neural Ordinary Differential Equations (Neural ODEs).

Below, we import our standard libraries. In this tutorial, we will use PyTorch Lightning. Additionally, we will use the ODE solvers from Torchdiffeq. You don't need to use GPUs for this tutorial, you can run the entire codebase in a CPU.

```
[ ]: %matplotlib inline
import time
import logging
import statistics
from typing import Optional, List

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from mpl_toolkits.mplot3d import Axes3D
```

(continues on next page)

(continued from previous page)

```
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

import torch
import torch.nn as nn
import torch.utils.data as data
import torch.nn.functional as F
from torch.utils.data import Dataset

try:
    import torchdiffeq
except ModuleNotFoundError:
    !pip install --quiet torchdiffeq
    import torchdiffeq

try:
    import rich
except ModuleNotFoundError:
    !pip install --quiet rich
    import rich

try:
    import pytorch_lightning as pl
except ModuleNotFoundError:
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from torchmetrics.classification import Accuracy

pl.seed_everything(42)

torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")

|| 229 kB 4.4 MB/s
|| 51 kB 5.0 MB/s

Global seed set to 42
Device: cpu
```

Dynamical Systems Primer

The most accurate definition of dynamical system is the following:

A dynamical system is a triple

$$(\mathcal{S}, \mathcal{T}, \Phi)$$

where - \mathcal{S} is the *state space* - \mathcal{T} is the *parameter space*, and - $\Phi : (\mathcal{T} \times \mathcal{S}) \longrightarrow \mathcal{S}$ is the evolution.

Some notes: - This is a very general definition that includes all sort of dynamical systems that you might encounter.
- In this tutorial we deal with ODEs where Φ plays the role of the *general solution*: indeed a 1-parameter family of transformations of the state space. $\mathcal{T} = \mathbb{R}_+$ is the time, and usually, $\mathcal{S} = \mathbb{R}^n$ is the state space. The evolution takes a point in space (initial value), a point in time, and returns the a point in space. This is the concept of a *flow*.

4.41.2 1. Differential Equations

We will deal with *initial value problems* (IVP) defined by a first-order ODE, and an initial value:

$$\dot{y} = f(y, t), \quad y(t_0) = y_0,$$

where we use the shorthand notation $\dot{y} := \frac{dy}{dt}$ common in physics.

A general solution to an ODE is a function $y : I \times \mathbb{R}^n \mathbb{R}^n$: a 1-parameter (usually time is the parameter) family of transformations of the state space. A 1-parameter family of transformations is often called a *flow*. The existence and uniqueness of solutions to an IVP is ensured by the Picard-Lindelöf theorem, provided the RHS of the ODE is *Lipschitz continuous*. Lipschitz continuity is a property that pops up quite often in ODE-related results in ML so we provide a definition here:

A function $f : X \subset \mathbb{R}^n \mathbb{R}^n$ is called *Lipschitz continuous* (with constant λ) if

$$\|f(x_1) - f(x_2)\| \leq \lambda \|x_1 - x_2\| \quad \forall x_1, x_2 \in X.$$

Note that this is a stronger condition than just continuity.

The rest of the tutorial is dedicated to numerical integration methods for finding solutions to IVPs.

Euler method

Solving differential equations analytically is not an option for complicated f s, given, for example, by neural networks. We need numerical solvers. Runge-Kutta methods are a family of iterative methods that find approximate solutions to IVPs. We will start with the simplest and most intuitive Runge-Kutta method, the **Euler** method.

Consider the IVP

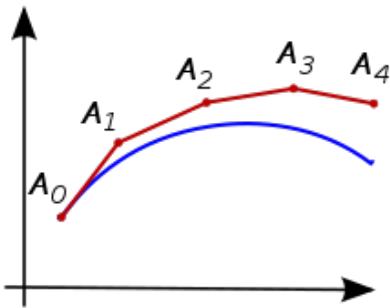
$$\dot{y} = f(y, t), \quad y(t_0) = y_0.$$

where $y(t_0)$, and f are given.

Pick a step-size $h > 0$, a number of steps N , and define

$$\begin{aligned} y_{n+1} &= y_n + h f(y_n, t_n) \\ t_{n+1} &= t_n + h. \end{aligned}$$

This is the most basic numerical integrator. One intuition behind the Euler method is that we are evolving the trajectories by iteratively taking small steps in the direction of the slope.



(Figure credit: Wikipedia)

Derivation

The mathematical explanation behind that intuition can be derived by taking the forward finite difference formula for the derivative:

$$\dot{y}(t_0) = \lim_{h \rightarrow 0} \frac{y(t_0 + h) - y(t_0)}{h} \approx \frac{y(t_0 + h) - y(t_0)}{h}.$$

By rearranging the terms, we can derive the forward Euler method.

We can also explain how this method works by considering the Taylor expansion of the solution $y(t)$ around t_{n+1}

$$y(t_{n+1}) = y(t_n) + h \frac{dy}{dt} \Big|_{t_n} + O(h^2),$$

and using the fact that $\frac{dy}{dt} \Big|_{t_n} = \dot{y}_{t_n} = f(y_n, t_n)$. We are left with

$$y(t_{n+1}) = y(t_n) + h f(y_n, t_n) + O(h^2),$$

which is precisely the Euler method step above.

Runge-Kutta Methods

Euler method is the simplest numerical integrator in a family of methods known as Runge-Kutta methods. Here we describe the RK4 method.

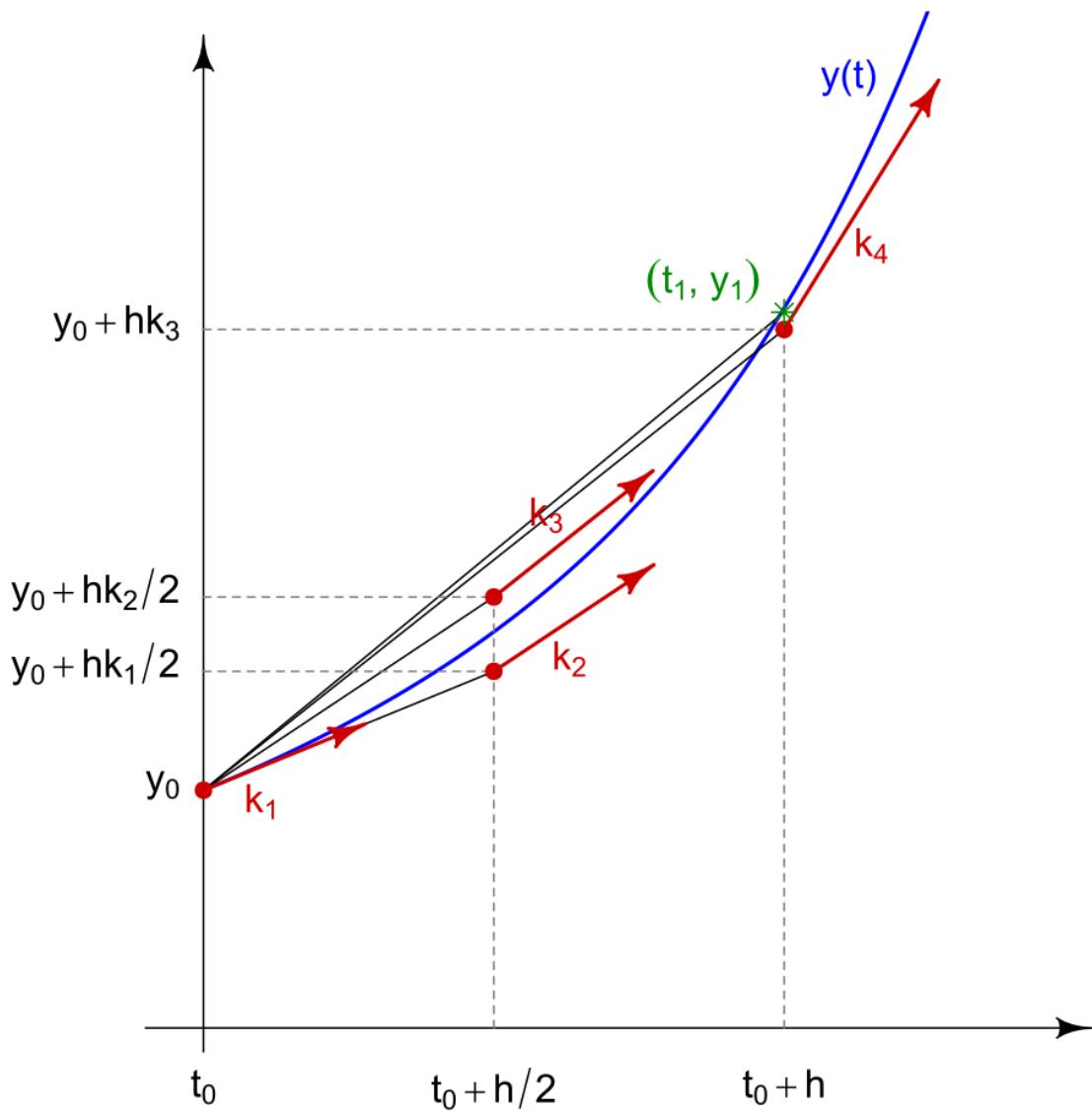
Consider the same IVP as above. Again, pick a step-size $h > 0$, a number of steps N , and define

$$\begin{aligned} y_{n+1} &= y_n + \frac{1}{6} h(k_1 + 2k_2 + 2k_3 + k_4) \\ t_{n+1} &= t_n + h \end{aligned}$$

for $n = 0, 1, 2, 3, \dots, N$ with

$$\begin{aligned} k_1 &= f(y_n, t_n) \\ k_2 &= f\left(y_n + h \frac{k_1}{2}, t_n + \frac{h}{2}\right) \\ k_3 &= f\left(y_n + h \frac{k_2}{2}, t_n + \frac{h}{2}\right) \\ k_4 &= f(y_n + h k_3, t_n + h). \end{aligned}$$

NB: As you probably noticed, Euler method is just RK4 but considerig only k_1 .



(Figure credit: Wikipedia)

Intuition

Here we give an intuitive explanation of why this method approximates solutions to IVPs. As with the Euler method, we will use the Taylor expansion of the solution.

Let $y(t)$ be the solution. Let us write the Taylor expansion of $y(t)$ in the neighborhood of t_n to the h^2 term:

$$y(t_{n+1}) = y(t_n) + h \frac{dy}{dt} \Big|_{t_n} + \frac{h^2}{2} \frac{d^2y}{dt^2} \Big|_{t_n} + O(h^3).$$

we know that $\frac{dy}{dt} \Big|_{t_n} = f(y_n, t_n)$ and therefore

$$\frac{d^2y}{dt^2} \Big|_{t_n} = \frac{df(y, t)}{dt} \Big|_{t_n} = \frac{\partial f}{\partial t} \Big|_{t_n} + f \frac{\partial f}{\partial y} \Big|_{t_n},$$

where we used the chain rule. The Taylor expansion becomes

$$y(t_{n+1}) = y(t_n) + hf(y_n, t_n) + \frac{h^2}{2} \left[\frac{\partial f}{\partial t} \Big|_{t_n} + f \frac{\partial f}{\partial y} \Big|_{t_n} \right] + O(h^3).$$

If we look at k_2 we can Taylor expand it correctly to $O(h^3)$ as

$$k_2 = f(y_n + \beta k_1, t_n + \alpha h) = h \left(f(y_n, t_n) + \alpha h \frac{\partial f}{\partial t} \Big|_{t_n} + \beta f \frac{\partial f}{\partial y} \Big|_{t_n} \right) + O(h^3),$$

which is precisely the third term in the Taylor expansion of $y(t_n)$. Comparing with the previous equation we find conditions on α and β . Note that the value of these coefficients depends on the order to which we decide to stop. Higher order coefficients can be computed in the same way.

Approximation errors

This method is called *classic Runge-Kutta* or *RK4*. This is a fourth-order method meaning that the *local truncation error* is of order $O(h^5)$, and the *total truncation error* is of order $O(h^4)$. Local truncation and total accumulation errors are defined as follows:

For a one-step integration method, such as RK4, of form

$$y_{n+1} = y_n + hA(y_n, t_n, h, f),$$

the *local truncation error* at time t_{n+1} , τ_{n+1} is

$$\tau_{n+1} = y(t_{n+1}) - y(t_n) - hA(y_n, t_n, h, f).$$

The *total truncation error* at time t_{n+1} , e_{n+1} is

$$e_{n+1} = y(t_{n+1}) - (y_0 + hA(y_0, t_0, h, f) + \dots + hA(y_n, t_n, h, f)).$$

Example: Lotka-Volterra equations

Consider the IVP

$$\begin{aligned}\dot{x} &= x - xy \\ \dot{y} &= xy - y,\end{aligned}$$

with initial value $(x_0, y_0) = (1, 2)$. There is no closed form solution to this system of ODEs. We compare performances of RK4 and simple Euler for different values of step-size h .

```
[ ]: def LV(x, y):
    return np.array([x - x*y, x*y - y])

def rk4(f, x0, y0, h, n):
    v = [0] * (n+1)
    v[0] = np.array([x0, y0])
```

(continues on next page)

(continued from previous page)

```

x = x0
y = y0
for i in range(1, n + 1):
    k1 = h*f(x, y)
    k2 = h*f(x + 0.5*k1[0], y + 0.5*k1[1])
    k3 = h*f(x + 0.5*k2[0], y + 0.5*k2[1])
    k4 = h*f(x + k3[0], y + k3[1])
    v[i] = v[i-1] + (k1 + k2 + k3 + k4)/6
    x = v[i][0]
    y = v[i][1]

t = np.array([i*h for i in range(0, n+1)])
return t, np.array(v)

def euler(f, x0, y0, h, n):

    v = [0] * (n+1)
    v[0] = np.array([x0, y0])
    x = x0
    y = y0

    for i in range(1, n + 1):
        v[i] = v[i-1] + h*f(x, y)
        x = v[i][0]
        y = v[i][1]

    t = np.array([i*h for i in range(0, n+1)])
    return t, np.array(v)

def plot_integrator(v_euler, v_rk4, t_euler, t_rk4, v_true, t_true, h):

    fig = plt.figure(figsize=(18,8))
    ax0 = fig.add_subplot(121)
    ax1 = fig.add_subplot(122)

    ax0.plot(t_euler, v_euler, marker = 'x')
    ax1.plot(t_rk4, v_rk4, marker = 'x')

    ax0.plot(t_true, v_true)
    ax1.plot(t_true, v_true)

    ax0.set_ylim(0, 3.5)
    ax1.set_ylim(0, 3.5)

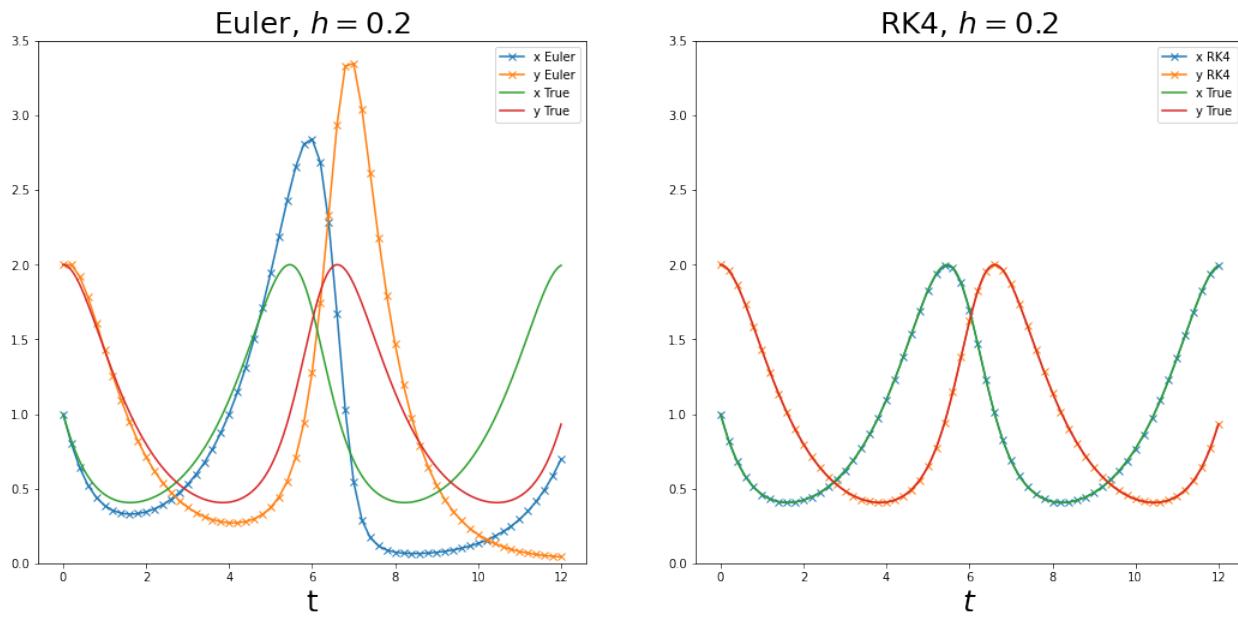
    ax0.set_xlabel(r"t", fontsize=25)
    ax0.set_title("Euler, $h=$"+h, fontsize=25)
    ax0.legend(["x Euler", "y Euler", "x True", "y True"])
    ax1.set_xlabel(r"$t$", fontsize=25)
    ax1.set_title("RK4, $h=$"+h, fontsize=25)
    ax1.legend(["x RK4", "y RK4", "x True", "y True"])

```

```
[ ]: h = 0.2
```

```
t_euler, v_euler = euler(LV, 1., 2., h, 60)
t_rk4, v_rk4 = rk4(LV, 1., 2., h, 60)
t_true, v_true = rk4(LV, 1., 2., 0.003, 4000)

plot_integrator(v_euler, v_rk4, t_euler, t_rk4, v_true, t_true, str(h))
```



Phase space and phase portrait

It is useful to introduce the concept of phase space and phase portrait. Let us consider the differential equation of the simple pendulum of mass 1, length 1, and g set to 1 for convenience:

$$\ddot{\theta} + \sin \theta = 0.$$

This is a second-order ODE, but it can be transformed in the following equivalent system of first-order ODEs by introducing the auxiliary variable $p_\theta = \dot{\theta}$:

$$\begin{aligned}\dot{\theta} &= p_\theta \\ \dot{p}_\theta &= -\sin(\theta)\end{aligned}\tag{4.100}$$

$S^1 \times \mathbb{R}$ is called the *phase space*, and $(\theta, p_\theta) \in S^1 \times \mathbb{R}$ are called *phase space variables* (θ is periodic and therefore lives in $S^1 = \{x \bmod 2\pi | x \in \mathbb{R}\}$).

Given a solution $(\theta(t), p_\theta(t))$ we can represent it as a path in the phase space $S^1 \times \mathbb{R}$:

```
[ ]: def pendulum(x, y):
    return np.array([y, -np.sin(x)])

def plot_phase_space(v):
    fig = plt.figure(figsize=(10, 6))
```

(continues on next page)

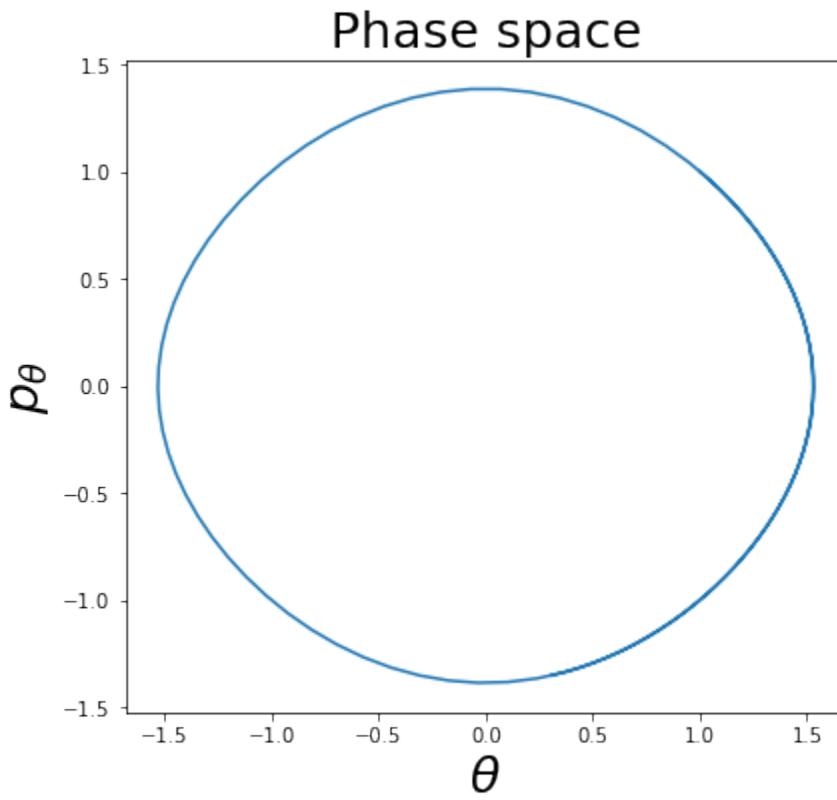
(continued from previous page)

```
ax0 = fig.add_subplot(111)

ax0.plot([p[0] for p in v], [p[1] for p in v])

ax0.set_xlabel(r"$\theta$", fontsize=25)
ax0.set_ylabel(r"$p_{\theta}$", fontsize=25)
ax0.set_title("Phase space", fontsize=25)
ax0.set_aspect('equal')
```

[]: t_pendulum, v_pendulum = rk4(pendulum, 1., 1., 0.1, 100)
plot_phase_space(v_pendulum)



A collection of paths in the phase space is called *phase space portrait* (or phase space diagram, or sometimes state space diagram), which can be thought as a geometric representation of the trajectories (here solutions of a system of ODEs) of a dynamical system in the phase plane. In a phase space portrait each initial conditions is represented by a different curve, or in the case of trivial solution (such as (0,0) in our example) a point.

[]: `def plot_phase_space_pendulum(V):`

```
fig = plt.figure(figsize=(18,8))
ax0 = fig.add_subplot(121)
for v in V:
    ax0.plot([p[0] for p in v], [p[1] for p in v], color='b')

ax0.set_xlabel(r"$\theta$", fontsize=25)
ax0.set_ylabel(r"$p_{\theta}$", fontsize=25)
```

(continues on next page)

(continued from previous page)

```

ax0.set_title("Phase space portrait", fontsize=25)
ax0.set_aspect('equal')

def plot_phase_space_LV(V):

    fig = plt.figure(figsize=(18,8))
    ax0 = fig.add_subplot(121)
    for v in V:
        ax0.plot([p[0] for p in v], [p[1] for p in v], color='b')

    ax0.set_xlabel(r"$x$", fontsize=25)
    ax0.set_ylabel(r"$y$", fontsize=25)
    ax0.set_title("Phase space portrait", fontsize=25)

```

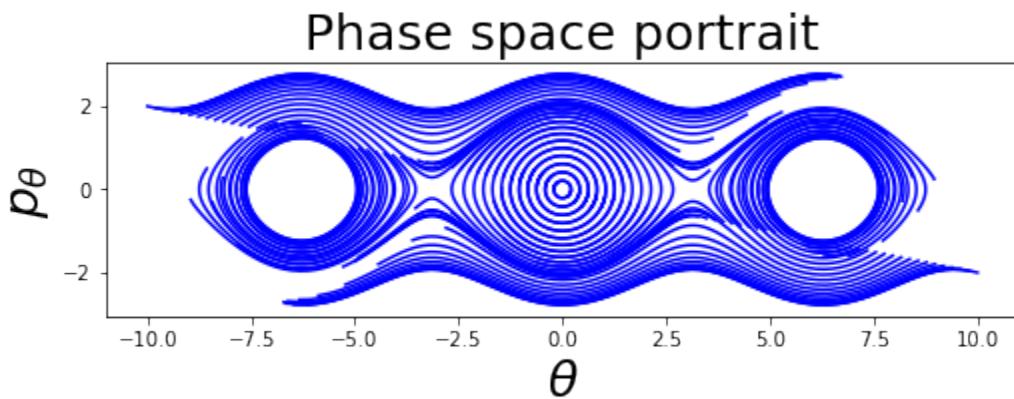
[]:

```

IV = [[5*i, -i] for i in np.linspace(-2, 2, 99)] # initial values
paths = []
for iv in IV:
    t, v = rk4(pendulum, iv[0], iv[1], 0.1, 70)
    paths.append(v)

plot_phase_space_pendulum(paths)

```



We can plot the phase space of the Lotka-Volterra system.

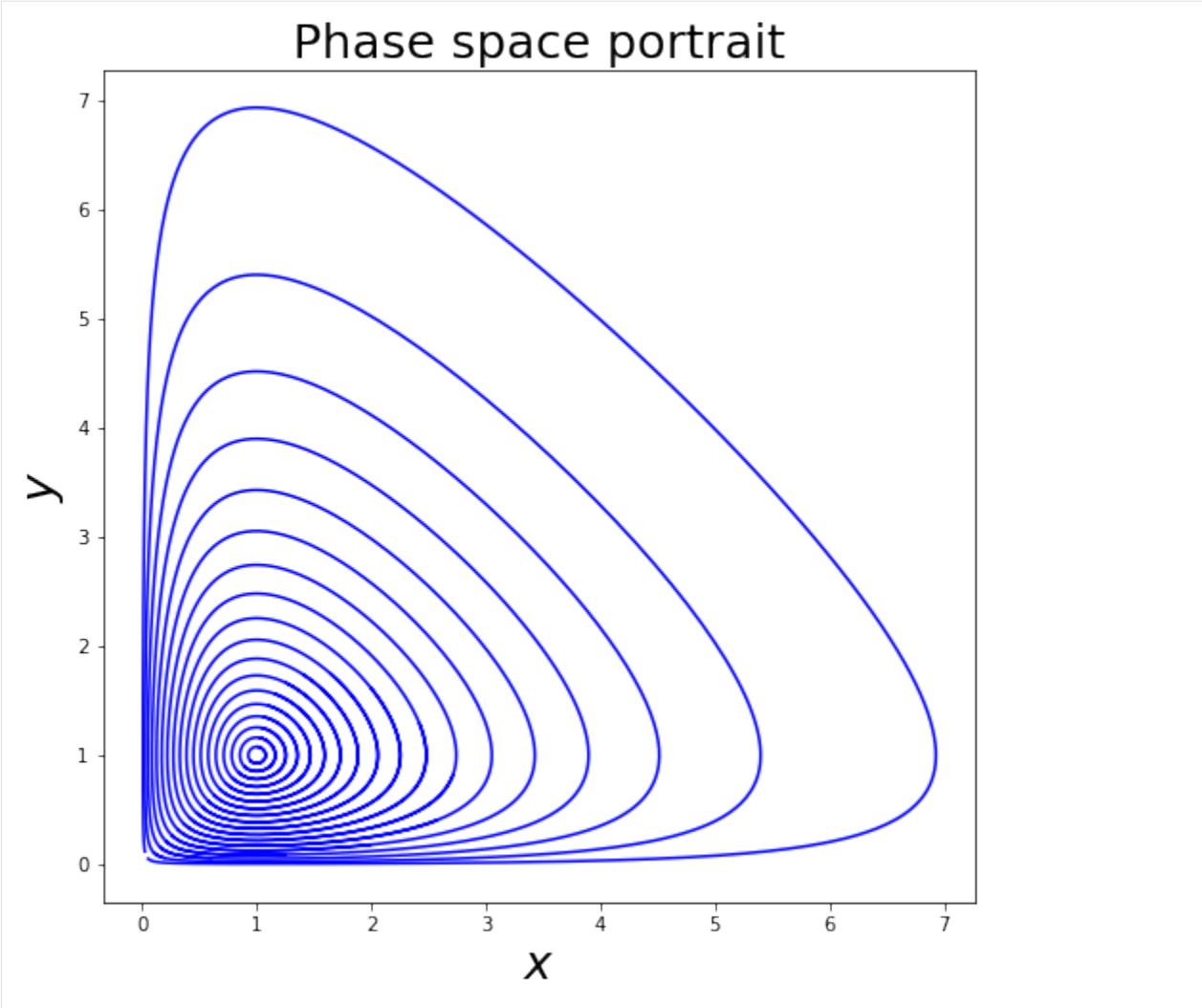
[]:

```

IV = [[i, i] for i in np.linspace(0, 1, 20)] # initial values
paths = []
for iv in IV:
    t, v = rk4(LV, iv[0], iv[1], 0.01, 1000)
    paths.append(v)

plot_phase_space_LV(paths)

```



4.41.3 2. Neural ODEs

Introduction

Given the intriguing properties of ODEs/solvers and the centuries-long literature on the topic, it seems intriguing to combine them with neural networks, i.e. try to model the transition function with a neural network. This combination could yield a powerful modelling tool, since neural networks are universal function approximators —they can in theory approximate any differentiable function to an arbitrary precision.

Remembering our earlier definitions, we have a first-order ODE:

$$\dot{\mathbf{y}}(t) = f(t, \mathbf{y}(t), \theta), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad f : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$$

Our goal is to solve initial value problems (IVP), i.e. predict $\mathbf{y}(t_1)$ given $\mathbf{y}(t_0)$.

$$\mathbf{y}(t_1) = \mathbf{y}(t_0) + \int_{t_0}^{t_1} f(\mathbf{y}(t), t, \theta) dt = \text{ODESolve}(\mathbf{y}(t_0), f, t_0, t_1, \theta)$$

We can use a numerical solver to perform the forward pass and solve the IVP. If we use, for example, Euler's method,

we have the following update rule:

$$\mathbf{y}(t+h) = \mathbf{y}(t) + h f(\mathbf{y}(t), t)$$

This formula looks almost identical to a ResNet block, and this was one of the main motivations for Neural ODEs.

Comparison to Resnets

Many popular deep learning architectures like ResNets² update hidden states by employing residual connections:

$$\mathbf{y}_{l+1} = \mathbf{y}_l + f(\mathbf{y}_l, \theta_l)$$

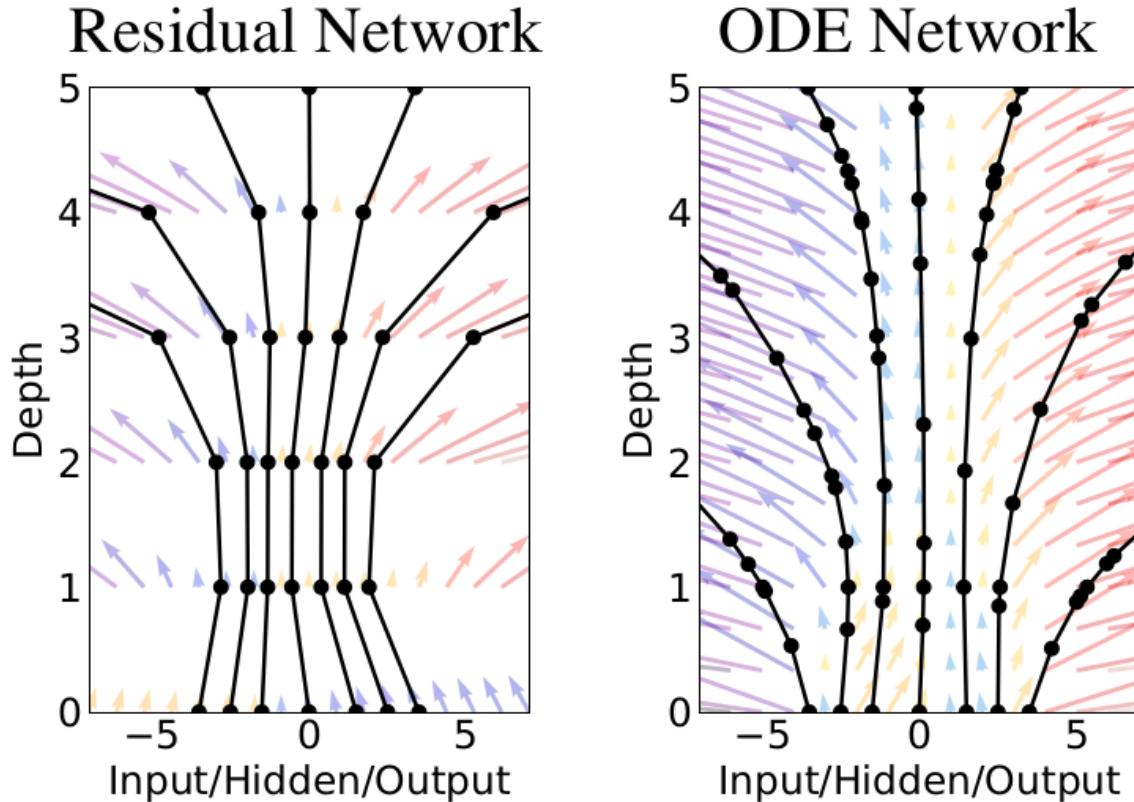
where f is a neural network with parameters θ_l , and \mathbf{y}_l and \mathbf{y}_{l+1} are the hidden states at subsequent layers, $l \in \{0, \dots, L\}$.

These updates can be seen as Euler discretizations of continuous transformations.

$$\dot{\mathbf{y}} = f(\mathbf{y}, t, \theta) \quad (4.102)$$

$$\begin{array}{c} \downarrow \\ \text{Euler Discretization} \\ \downarrow \\ \mathbf{y}_{n+1} = \mathbf{y}_n + h f(\mathbf{y}_n, t_n) \end{array} \quad (4.103)$$

Circling back to the continuous realm again, what happens in a residual network (with step sizes h) if we consider the continuous limit of each discrete layer in the network? In other words, what happens as we add more layers and take smaller steps? The answer seems rather astounding: instead of having a discrete number of layers between the input and output domains, we allow the evolution of the hidden states to become continuous!



(Figure credit: <https://arxiv.org/abs/1806.07366>)

² He, Kaiming, et al. “**Deep residual learning for image recognition**”. CVPR 2016. <https://arxiv.org/abs/1512.03385>

Backpropagation

We now have a way to perform the forward pass for our model. How do we backpropagate and train the network, though?

One very straightforward way to perform backprop is to back-propagate through the solver. This would work since the forward pass operations are continuous and differentiable. The problem, however, is that this incurs a high-memory cost and we would need to save all intermediate activations of the solver.

More importantly, though, our goal should be to try to approximate the exact derivative, rather than differentiating the approximation.

We want to optimize our scalar-valued loss function L with respect to the model parameters θ .

$$L(\mathbf{x}(t_1)) = L\left(\mathbf{x}(t_0) + \int_{t_0}^{t_1} f(\mathbf{x}(t), t, \theta) dt\right) = L(\text{ODESolve}(\mathbf{x}(t_0), f, t_0, t_1, \theta))$$

To optimize L we require gradients with respect to θ . The problem is to efficiently calculate $\frac{dL(\mathbf{x}(t_1))}{d\theta}$ without storing all the function activations from the forward pass. **Adjoint method to the rescue!** The adjoint sensitivity method was developed in 1962 by Pontryagin et al.¹ It leverages the fact that the forward pass is the solution to an ODE, and computes gradients by solving a second, augmented ODE backwards in time.

Similar to standard neural networks, we start with determining how the gradient of the loss depends on the hidden state. This quantity is called the *adjoint* $\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{x}(t)}$. It satisfies the following IVP:

$$\dot{\mathbf{a}}(t) = -\mathbf{a}(t)^\top \frac{\partial f(\mathbf{x}(t), t, \theta)}{\partial \mathbf{x}}, \quad \mathbf{a}(t_1) = \frac{\partial L}{\partial \mathbf{x}(t_1)}.$$

Thus, starting from the initial (remember we are running backwards) value $\mathbf{a}(t_1) = \frac{\partial L}{\partial \mathbf{x}(t_1)}$, we can compute $\mathbf{a}(t_0) = \frac{\partial L}{\partial \mathbf{x}(t_0)}$ by another call to an ODE solver.

Finally, computing the gradients with respect to the parameters θ requires evaluating a third integral, which depends on both $\mathbf{x}(t)$ and $\mathbf{a}(t)$:

$$\frac{dL}{d\theta} = - \int_{t_1}^{t_0} \mathbf{a}(t)^\top \frac{\partial f}{\partial \theta} dt,$$

So this method trades off computation for memory – in fact the memory requirement for this gradient calculation is $\mathcal{O}(1)$ with respect to the number of layers!

Here you can find a very good explanation of the following result based on Lagrange multipliers:

¹ Pontryagin, L.S. et al. “**The mathematical theory of optimal processes**”. 1962

The full algorithm for *reverse mode auto-differentiation* is as follows:

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

```

Input: dynamics parameters  $\theta$ , start time  $t_0$ , stop time  $t_1$ , final state  $\mathbf{z}(t_1)$ , loss gradient  $\frac{\partial L}{\partial \mathbf{z}(t_1)}$ 
 $s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$                                 ▷ Define initial augmented state
def aug_dynamics([ $\mathbf{z}(t)$ ,  $\mathbf{a}(t)$ ,  $\cdot$ ],  $t$ ,  $\theta$ ):          ▷ Define dynamics on augmented state
    return [ $f(\mathbf{z}(t), t, \theta)$ ,  $-\mathbf{a}(t)^\top \frac{\partial f}{\partial \mathbf{z}}$ ,  $-\mathbf{a}(t)^\top \frac{\partial f}{\partial \theta}$ ]      ▷ Compute vector-Jacobian products
    [ $\mathbf{z}(t_0)$ ,  $\frac{\partial L}{\partial \mathbf{z}(t_0)}$ ,  $\frac{\partial L}{\partial \theta}$ ] = ODESolve( $s_0$ , aug_dynamics,  $t_1$ ,  $t_0$ ,  $\theta$ )      ▷ Solve reverse-time ODE
return  $\frac{\partial L}{\partial \mathbf{z}(t_0)}$ ,  $\frac{\partial L}{\partial \theta}$                                               ▷ Return gradients

```

(Figure credit: <https://arxiv.org/abs/1806.07366>)

Time to program some Neural ODEs

In this tutorial we will be working with Half Moons Dataset, a non-linearly separable, binary classification dataset.

This tutorial has been based on the excellent TorchDyn tutorials (<https://github.com/DiffEqML/torchdyn>), as well as the original TorchDiffEq examples (<https://github.com/rtqichen/torchdiffeq>).

```
[ ]: class MoonsDataset(Dataset):
    """Half Moons Classification Dataset

    Adapted from https://github.com/DiffEqML/torchdyn
    """

    def __init__(self, num_samples=100, noise_std=1e-4):
        self.num_samples = num_samples
        self.noise_std = noise_std
        self.X, self.y = self.generate_moons(num_samples, noise_std)

    @staticmethod
    def generate_moons(num_samples=100, noise_std=1e-4):
        """Creates a *moons* dataset of `num_samples` data points.
        :param num_samples: number of data points in the generated dataset
        :type num_samples: int
        :param noise_std: standard deviation of noise magnitude added to each data point
        :type noise_std: float
        """
        num_samples_out = num_samples // 2
        num_samples_in = num_samples - num_samples_out
        theta_out = np.linspace(0, np.pi, num_samples_out)
        theta_in = np.linspace(0, np.pi, num_samples_in)
        outer_circ_x = np.cos(theta_out)
        outer_circ_y = np.sin(theta_out)
        inner_circ_x = 1 - np.cos(theta_in)
        inner_circ_y = 1 - np.sin(theta_in) - 0.5

        X = np.vstack([np.append(outer_circ_x, inner_circ_x),
                      np.append(outer_circ_y, inner_circ_y)]).T
        y = np.hstack([np.zeros(num_samples_out), np.ones(num_samples_in)])

        if noise_std is not None:
            X += noise_std * np.random.rand(num_samples, 2)

        X = torch.Tensor(X)
        y = torch.LongTensor(y)
        return X, y

    def __len__(self):
        return self.num_samples

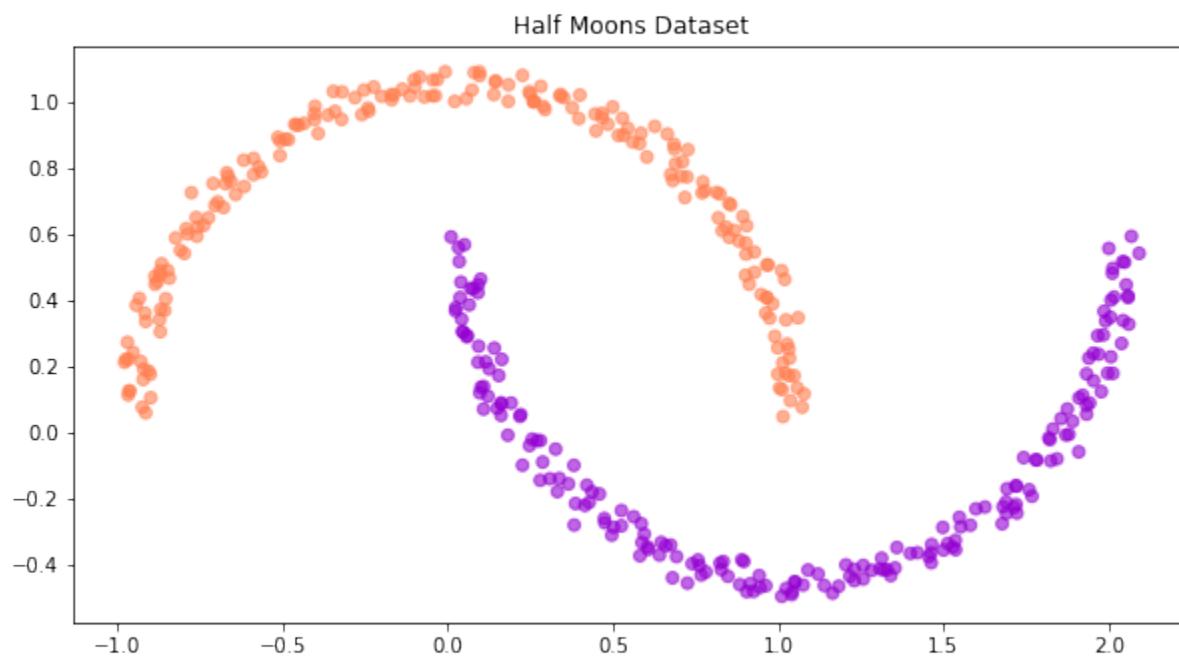
    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

```
[ ]: def plot_binary_classification_dataset(X, y, title=None):
    CLASS_COLORS = ['coral', 'darkviolet']
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.scatter(X[:, 0], X[:, 1], color=[CLASS_COLORS[yi.int()] for yi in y], alpha=0.6)
    ax.set_aspect('equal')
    if title is not None:
        ax.set_title(title)

    return fig, ax
```

Let's create a sample dataset and visualize it.

```
[ ]: sample_dataset = MoonsDataset(num_samples=400, noise_std=1e-1)
fig, ax = plot_binary_classification_dataset(sample_dataset.X, sample_dataset.y, title=
    'Half Moons Dataset')
```



Let's now create the train, validation, and test sets, with their corresponding data loaders. We will create a single big dataset and randomly split it in train, val, and test sets.

```
[ ]: def split_dataset(dataset_size:int, split_percentages:List[float]) -> List[int]:
    split_sizes = [int(pi * dataset_size) for pi in split_percentages]
    split_sizes[0] += dataset_size - sum(split_sizes)
    return split_sizes

class ToyDataModule(pl.LightningDataModule):
    def __init__(self, dataset_size:int, split_percentages:Optional[float]=None):
        super().__init__()
        self.dataset_size = dataset_size
        if split_percentages is None:
            split_percentages = [0.8, 0.1, 0.1]
```

(continues on next page)

(continued from previous page)

```

    self.split_sizes = split_dataset(self.dataset_size, split_percentages)

    def prepare_data(self):
        pass

    def setup(self, stage: Optional[str] = None):
        pass

    def train_dataloader(self):
        train_loader = torch.utils.data.DataLoader(self.train_set, batch_size=len(self.
        ↪train_set), shuffle=True)
        return train_loader

    def val_dataloader(self):
        val_loader = torch.utils.data.DataLoader(self.val_set, batch_size=len(self.val_.
        ↪set), shuffle=False)
        return val_loader

    def test_dataloader(self):
        test_loader = torch.utils.data.DataLoader(self.test_set, batch_size=len(self.
        ↪test_set), shuffle=False)
        return test_loader

class HalfMoonsDataModule(ToyDataModule):
    def __init__(self, dataset_size:int, split_percentages:Optional[float]=None):
        super().__init__(dataset_size, split_percentages)

    def setup(self, stage: Optional[str] = None):
        dataset = MoonsDataset(num_samples=self.dataset_size, noise_std=1e-1)
        self.train_set, self.val_set, self.test_set = torch.utils.data.random_
        ↪split(dataset, self.split_sizes)

```

Now we define the core of our Neural ODE model.

```

[ ]: class _ODEFunc(nn.Module):
    def __init__(self, module, autonomous=True):
        super().__init__()
        self.module = module
        self.autonomous = autonomous

    def forward(self, t, x):
        if not self.autonomous:
            x = torch.cat([torch.ones_like(x[:, [0]]) * t, x], 1)
        return self.module(x)

class ODEBlock(nn.Module):
    def __init__(self, odefunc: nn.Module, solver: str = 'dopri5',
                 rtol: float = 1e-4, atol: float = 1e-4, adjoint: bool = True,
                 autonomous: bool = True):
        super().__init__()

```

(continues on next page)

(continued from previous page)

```

self.odefunc = _ODEFunc(odefunc, autonomous=autonomous)
self.rtol = rtol
self.atol = atol
self.solver = solver
self.use_adjoint = adjoint
self.integration_time = torch.tensor([0, 1], dtype=torch.float32)

@property
def ode_method(self):
    return torchdiffeq.odeint_adjoint if self.use_adjoint else torchdiffeq.odeint

def forward(self, x: torch.Tensor, adjoint: bool = True, integration_time=None):
    integration_time = self.integration_time if integration_time is None else_
    integration_time
    integration_time = integration_time.to(x.device)
    ode_method = torchdiffeq.odeint_adjoint if adjoint else torchdiffeq.odeint
    out = ode_method(
        self.odefunc, x, integration_time, rtol=self.rtol,
        atol=self.atol, method=self.solver)
    return out

```

We will wrap everything together in a **LightningModule**.

```

[ ]: class Learner(pl.LightningModule):
    def __init__(self, model:nn.Module, t_span:torch.Tensor, learning_rate:float=5e-3):
        super().__init__()
        self.model = model
        self.t_span = t_span
        self.learning_rate = learning_rate
        self.accuracy = Accuracy(num_classes=2)

    def forward(self, x):
        return self.model(x)

    def inference(self, x, time_span):
        return self.model(x, adjoint=False, integration_time=time_span)

    def inference_no_projection(self, x, time_span):
        return self.model.forward_no_projection(x, adjoint=False, integration_time=time_
        span)

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_pred = self(x)
        y_pred = y_pred[-1] # select last point of solution trajectory
        loss = nn.CrossEntropyLoss()(y_pred, y)
        self.log('train_loss', loss, prog_bar=True, logger=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_pred = self(x)

```

(continues on next page)

(continued from previous page)

```

y_pred = y_pred[-1] # select last point of solution trajectory
loss = nn.CrossEntropyLoss()(y_pred, y)
self.log('val_loss', loss, prog_bar=True, logger=True)
acc = self.accuracy(y_pred.softmax(dim=-1), y)
self.log('val_accuracy', acc, prog_bar=True, logger=True)
return loss

def test_step(self, batch, batch_idx):
    x, y = batch
    y_pred = self(x)
    y_pred = y_pred[-1] # select last point of solution trajectory
    loss = nn.CrossEntropyLoss()(y_pred, y)
    self.log('test_loss', loss, prog_bar=True, logger=True)
    acc = self.accuracy(y_pred.softmax(dim=-1), y)
    self.log('test_accuracy', acc, prog_bar=True, logger=True)
    return loss

def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.model.parameters(), lr=self.learning_rate)
    return optimizer

```

Finally, it is time to actually define a Neural ODE and train it. We will use a simple 2-layer MLP with a *tanh* activation and 64 hidden dimensions. We will train the model using the adjoint method for backpropagation.

A quick note on the architectural choices for our model. As mentioned in the first part of this tutorial, the **Picard-Lindelöf theorem** (Coddington and Levinson, 1955) states that the solution to an initial value problem **exists and is unique** if the differential equation is *uniformly Lipschitz continuous* in \mathbf{z} and *continuous* in t . It turns out that this theorem holds for our model if the neural network has finite weights and uses Lipschitz nonlinearities, such as tanh or relu. However, not all tools are our deep learning arsenal is Lipschitz. For example, as shown in [**The Lipschitz Constant of Self-Attention**](#) by Hyunjik Kim et al., standard self-attention is ***not*** Lipschitz. The authors propose alternative forms of self-attention that are Lipschitz.

```

[ ]: adjoint = True
data_module = HalfMoonsDataModule(1000)
t_span = torch.linspace(0, 1, 2)
f = nn.Sequential(
    nn.Linear(2, 64),
    nn.Tanh(),
    nn.Linear(64, 2))
model = ODEBlock(f, adjoint=adjoint)
learner = Learner(model, t_span)

trainer = pl.Trainer(
    max_epochs=200,
    accelerator="gpu" if torch.cuda.is_available() else "cpu",
    devices=1,
    callbacks=[
        pl.callbacks.ModelCheckpoint(mode="max", monitor="val_accuracy"),
        pl.callbacks.RichProgressBar(),
    ],
    log_every_n_steps=1,
)
trainer.fit(learner, datamodule=data_module)

```

(continues on next page)

(continued from previous page)

```
val_result = trainer.validate(learner, datamodule=data_module, verbose=True)
test_result = trainer.test(learner, datamodule=data_module, verbose=True)
```

GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
Missing logger folder: /content/lightning_logs

Name	Type	Params
0	model	ODEBlock
1	accuracy	Accuracy

Trainable params: 322
Non-trainable params: 0
Total params: 322
Total estimated model params size (MB): 0

Output()

Output()

Validate metric DataLoader 0

val_accuracy	1.0
val_loss	0.0021277053747326136

Output()

Test metric DataLoader 0

test_accuracy	1.0
test_loss	0.0018281997181475163

Excellent! It seems that in less than 200 epochs we have achieved perfect validation accuracy. Let's now use the trained model to run inference and visualize the trajectories using a dense time span of 100 timesteps.

```
[ ]: @torch.no_grad()
def run_inference(learner, data_loader, time_span):
    learner.to(device)
    trajectories = []
    classes = []
    time_span = torch.from_numpy(time_span).to(device)
    for data, target in data_loader:
        data = data.to(device)
        traj = learner.inference(data, time_span).cpu().numpy()
        trajectories.append(traj)
        classes.extend(target.numpy())
    trajectories = np.concatenate(trajectories, 1)
    return trajectories, classes

time_span = np.linspace(0.0, 1.0, 100)
trajectories, classes = run_inference(learner, data_module.train_dataloader(), time_span)

colors = ['coral', 'darkviolet']
class_colors = [colors[ci] for ci in classes]
```

We will now define a few functions to visualize the learned trajectories, the state-space, and the learned vector field.

```
[1]: #@title You can omit reading this piece of code.

def plot_trajectories(time_span, trajectories, class_colors):
    fig = plt.figure(figsize=(12,6))
    ax0 = fig.add_subplot(121)
    ax1 = fig.add_subplot(122)
    for i in range(trajectories.shape[1]):
        ax0.plot(time_span, trajectories[:, i, 0], color=class_colors[i], alpha=0.1)
        ax1.plot(time_span, trajectories[:, i, 1], color=class_colors[i], alpha=0.1)

    ax0.set_xlabel(r"$t$ [Depth]")
    ax0.set_ylabel(r"$\mathbf{z}_0(t)$")
    ax0.set_title("Dimension 0")
    ax1.set_xlabel(r"$t$ [Depth]")
    ax1.set_ylabel(r"$\mathbf{z}_1(t)$")
    ax1.set_title("Dimension 1")

def plot_trajectories_3d(time_span, trajectories, class_colors):
    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(111, projection='3d')
    for i in range(trajectories.shape[1]):
        ax.plot(trajectories[:, i, 0], trajectories[:, i, 1], time_span,
                color=class_colors[i], alpha=0.1)

    ax.set_title('3D Trajectories')
    ax.set_xlabel(r"$\mathbf{z}_0(t)$")
    ax.set_ylabel(r"$\mathbf{z}_1(t)$")
    ax.set_zlabel(r"$t$")
```

(continues on next page)

(continued from previous page)

```

def plot_trajectories_animation(time_span, trajectories, colors, classes, lim=10.0):
    def animate_frame(t):
        ax.cla()
        ax.set_xlim(-lim, lim)
        ax.set_ylim(-lim, lim)
        ax.set_title('Trajectories')
        ax.set_xlabel(r"$\mathbf{z}_0(t)$")
        ax.set_ylabel(r"$\mathbf{z}_1(t)$")

        zero_classes = np.array(classes) == 0
        one_classes = np.array(classes) == 1

        scatter_zero = ax.plot(
            trajectories[t, zero_classes, 0], trajectories[t, zero_classes, 1],
            'o', color=colors[0], alpha=0.2+0.8*t/len(time_span))
        scatter_one = ax.plot(
            trajectories[t, one_classes, 0], trajectories[t, one_classes, 1],
            'o', color=colors[1], alpha=0.2+0.8*t/len(time_span))
        return scatter_zero, scatter_one

    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(111)
    anim = FuncAnimation(fig, animate_frame, frames=len(time_span))
    plt.close(fig)
    return anim

def plot_augmented_trajectories_animation(time_span, trajectories, colors, classes, lim=10.0):
    def animate_frame(t):
        ax.cla()
        ax.set_xlim(-lim, lim)
        ax.set_ylim(-lim, lim)
        ax.set_zlim(-lim, lim)
        ax.set_title('Trajectories')
        ax.set_xlabel(r"$\mathbf{z}_0(t)$")
        ax.set_ylabel(r"$\mathbf{z}_1(t)$")
        ax.set_zlabel(r"$\mathbf{z}_2(t)$")

        zero_classes = np.array(classes) == 0
        one_classes = np.array(classes) == 1

        scatter_zero = ax.plot(
            trajectories[t, zero_classes, 0], trajectories[t, zero_classes, 1],
            trajectories[t, zero_classes, 2],
            'o', color=colors[0], alpha=0.2+0.8*t/len(time_span))
        scatter_one = ax.plot(
            trajectories[t, one_classes, 0], trajectories[t, one_classes, 1],
            trajectories[t, one_classes, 2],
            'o', color=colors[1], alpha=0.2+0.8*t/len(time_span))

```

(continues on next page)

(continued from previous page)

```

    return scatter_zero, scatter_one

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
anim = FuncAnimation(fig, animate_frame, frames=len(time_span))
plt.close(fig)
return anim

def plot_state_space(trajectories, class_colors, ax=None):
    if ax is None:
        fig = plt.figure(figsize=(8, 8))
        ax = fig.add_subplot(111)

    for i in range(trajectories.shape[1]):
        ax.plot(trajectories[:, i, 0], trajectories[:, i, 1],
                color=class_colors[i], alpha=0.1)

    ax.set_title('State-Space Diagram')
    ax.set_xlabel(r"$x$")
    ax.set_ylabel(r"$y$")

def plot_augmented_state_space(trajectories, class_colors, ax=None):
    if ax is None:
        fig = plt.figure(figsize=(8, 8))
        ax = fig.add_subplot(111, projection='3d')

    for i in range(trajectories.shape[1]):
        ax.plot(trajectories[:, i, 0], trajectories[:, i, 1], trajectories[:, i, 2],
                color=class_colors[i], alpha=0.1)

    ax.set_title('State-Space Diagram')
    ax.set_xlabel(r"$x$")
    ax.set_ylabel(r"$y$")
    ax.set_zlabel(r"$z$")

def plot_static_vector_field(model, trajectory, N=50, device='cpu', ax=None):
    X, Y = np.mgrid[trajectory[:, 0].min():trajectory[:, 0].max():N*1j,
                    trajectory[:, 1].min():trajectory[:, 1].max():N*1j]
    X = X.T
    Y = Y.T
    P = np.vstack([X.ravel(), Y.ravel()]).T
    P = torch.Tensor(P).to(device)

    with torch.no_grad():
        vector_field = model.odefunc(0.0, P).cpu()
    vector_norm = vector_field.norm(dim=1).view(N, N).numpy()

    vector_field = vector_field.view(N, N, 2).numpy()

```

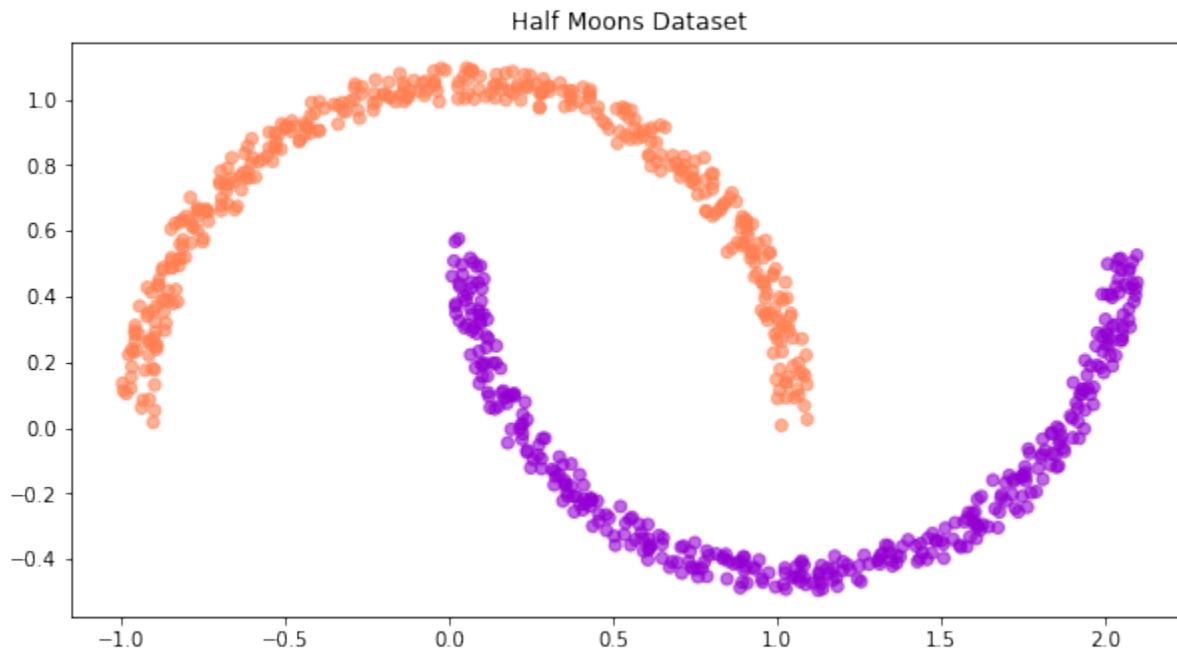
(continues on next page)

(continued from previous page)

```
if ax is None:  
    fig = plt.figure(figsize=(8, 8))  
    ax = fig.add_subplot(111)  
ax.contourf(X, Y, vector_norm, cmap='RdYlBu')  
ax.streamplot(X, Y, vector_field[:, :, 0], vector_field[:, :, 1], color='k')  
  
ax.set_xlim([X.min(), X.max()])  
ax.set_ylim([Y.min(), Y.max()])  
ax.set_xlabel(r"x")  
ax.set_ylabel(r"y")  
ax.set_title("Learned Vector Field")
```

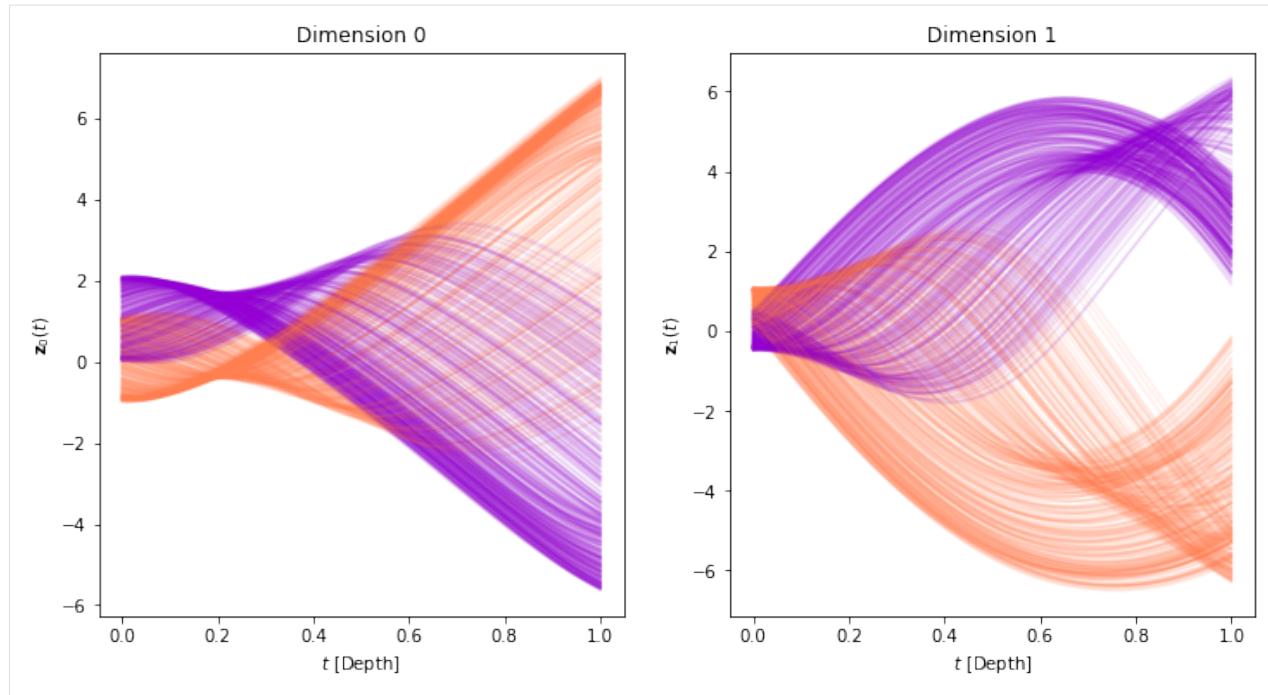
Before we visualize the trajectories, let's plot the (training) data once again:

```
[ ]: fig, ax = plot_binary_classification_dataset(*data_module.train_set[:, title='Half  
Moons Dataset')
```



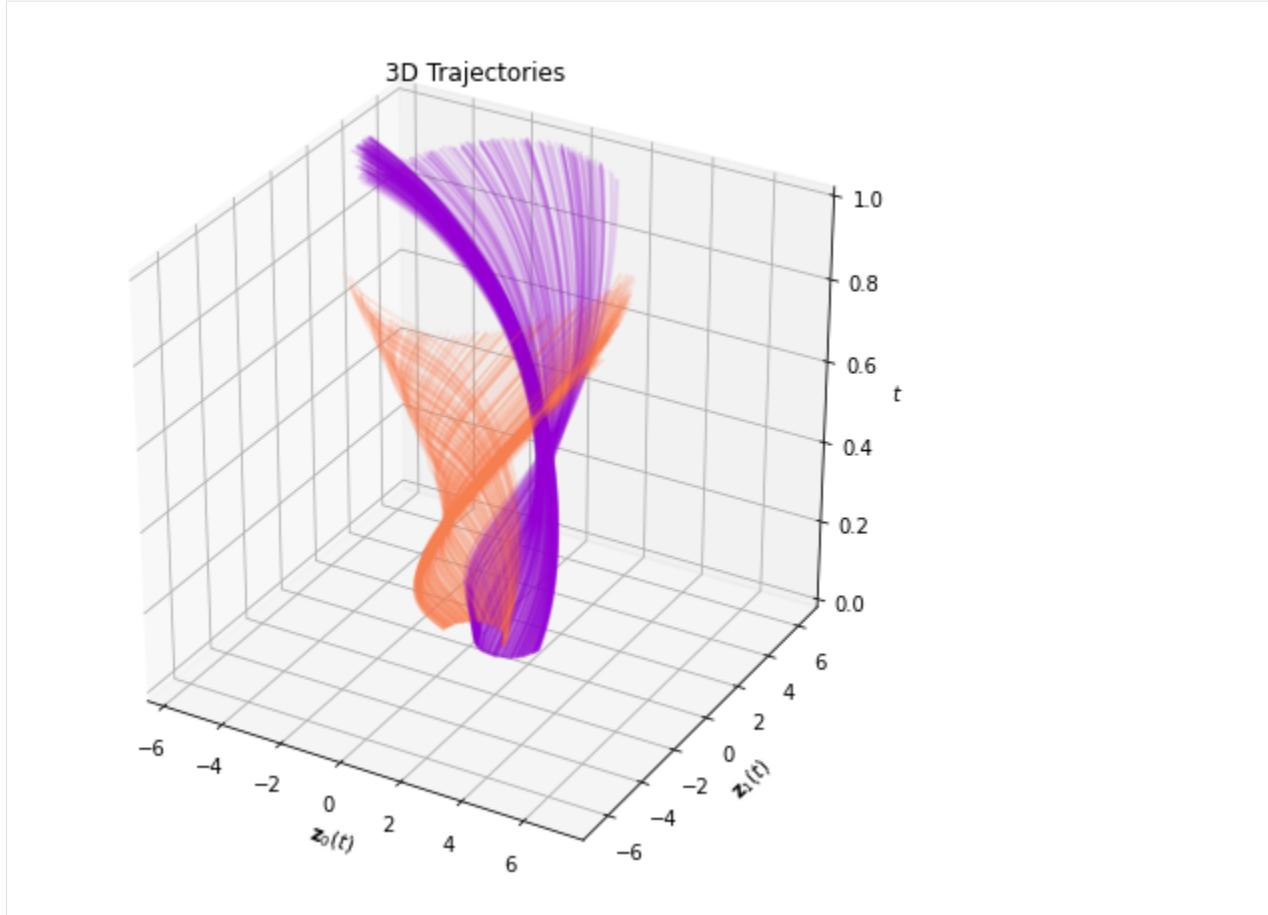
Below we visualize the evolution for each of the 2 inputs dimensions as a function of time (depth):

```
[ ]: plot_trajectories(time_span, trajectories, class_colors)
```



And the same evolution combined in a single plot:

```
[ ]: plot_trajectories_3d(time_span, trajectories, class_colors)
```

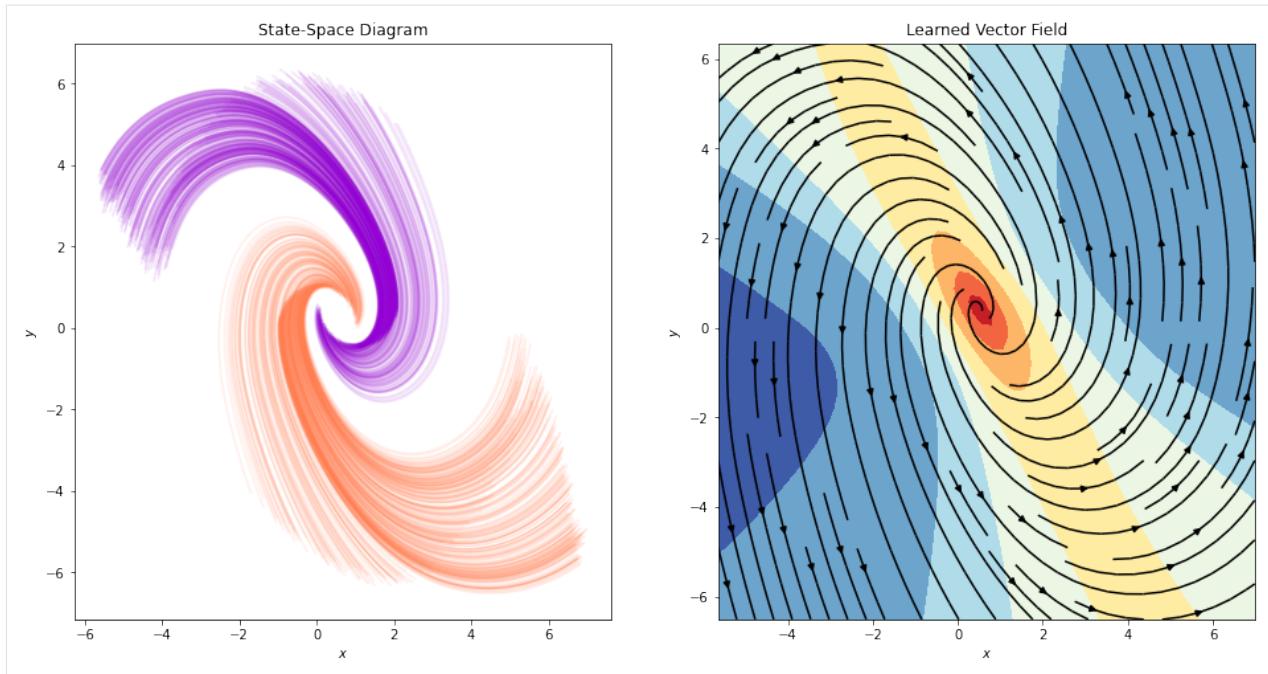


The 3D plot can be somewhat complicated to decipher. Thus, we also plot an animated version of the evolution. Each timestep of the animation is a slice on the temporal axis of the figure above.

```
[ ]: anim = plot_trajectories_animation(time_span, trajectories, colors, classes, lim=8.0)
HTML(anim.to_html5_video())
<IPython.core.display.HTML object>
```

Finally, let's visualize the state-space diagram and the learned vector field:

```
[ ]: fig, ax = plt.subplots(1, 2, figsize=(16, 8))
plot_state_space(trajectories, class_colors, ax=ax[0])
plot_static_vector_field(model, trajectories, ax=ax[1], device=device)
```



Neural ODEs can only describe homeomorphisms

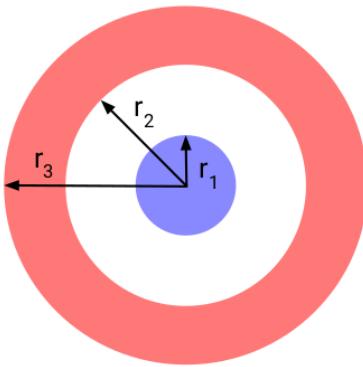
It seems that the network can indeed do a very good job at separating data from these 2 classes. We will now move to different setting, which appears similar at first in terms of difficulty, but will turn out to be very hard, even impossible theoretically for Neural ODEs to solve in their basic form.

Neural ODEs describe homeomorphisms (flows). Inputs/hidden states/outputs have the same dimensionality. They form non-intersecting trajectories.

Since Neural ODEs cannot model non-homeomorphisms, they cannot, for example, separate a 2d concentric annuli/circles dataset.

$$0 < r_1 < r_2 < r_3, \quad g : \mathbb{R}^2 \rightarrow \mathbb{R}$$

$$\begin{cases} g(\mathbf{x}) = 0, & \|\mathbf{x}\| \leq r_1, \\ g(\mathbf{x}) = 1, & r_2 \leq \|\mathbf{x}\| \leq r_3 \end{cases}$$



(Figure credit: Emilien Dupont et al.)

Neural ODEs cannot represent that function, the features of NODEs preserve the topology of the input space. NODEs can only continuously deform the input space, and cannot, for example, tear a connected region apart.

In practice, however, Neural ODEs are not trained on continuous regions of space, they are rather trained on a finite number of data points. This means that NODEs can “cheat” and stretch space, squeezing through the gaps between data points. This can lead to ill-posed ODE problems that are numerically expensive to solve.

Let's see this effect in practice. First, we will define a similar dataset that comprises 2 2D concentric circles.

```
[ ]: def rand_sphere(num_samples:int, dim:int, radius:float) -> torch.Tensor:
    """Uniform sample from a `dim`-dimensional sphere of radius `radius`"""
    :param num_samples: number of points to sample
    :type num_samples: int
    :param dim: dimension of the hyper-sphere
    :type dim: int
    :param radius: radius of the hyper-sphere
    :type radius: float
    """
    v = torch.randn(num_samples, dim)
    points = radius * F.normalize(v, dim=-1)
    return points

class ConcentricCircles(Dataset):
    """Concentric Circles Classification Dataset

    Adapted from https://github.com/DiffEqML/torchdyn
    """

    def __init__(self, num_samples=100, noise_std=1e-4, inner_radius=0.5,
                 outer_radius=1.0):
        self.num_samples = num_samples
        self.noise_std = noise_std
        self.X, self.y = self.generate_concentric_circles(num_samples, noise_std)

    @staticmethod
    def generate_concentric_circles(num_samples:int=100, noise_std:float=1e-4,
                                     inner_radius:float=0.5, outer_radius:int=1.0):
        """Creates a *concentric circles* dataset of `num_samples` datasets points.
        :param num_samples: number of datasets points in the generated dataset
        :type num_samples: int
        :param noise_std: standard deviation of noise magnitude added to each datasets
        ↪point
        :type noise_std: float
        :param inner_radius: radius of the inner circle
        :type inner_radius: float
        :param outer_radius: radius of the outer circle
        :type outer_radius: float
        """
        y = torch.zeros(num_samples, dtype=torch.long)
        y[:num_samples // 2] = 1

        X = torch.zeros((num_samples, 2))
        X[:num_samples // 2] = rand_sphere(num_samples // 2, 2, inner_radius)
        X[num_samples // 2:] = rand_sphere(num_samples - num_samples // 2, 2, outer_
        ↪radius)
```

(continues on next page)

(continued from previous page)

```
X += noise_std * torch.randn((num_samples, 2))

    return X, y

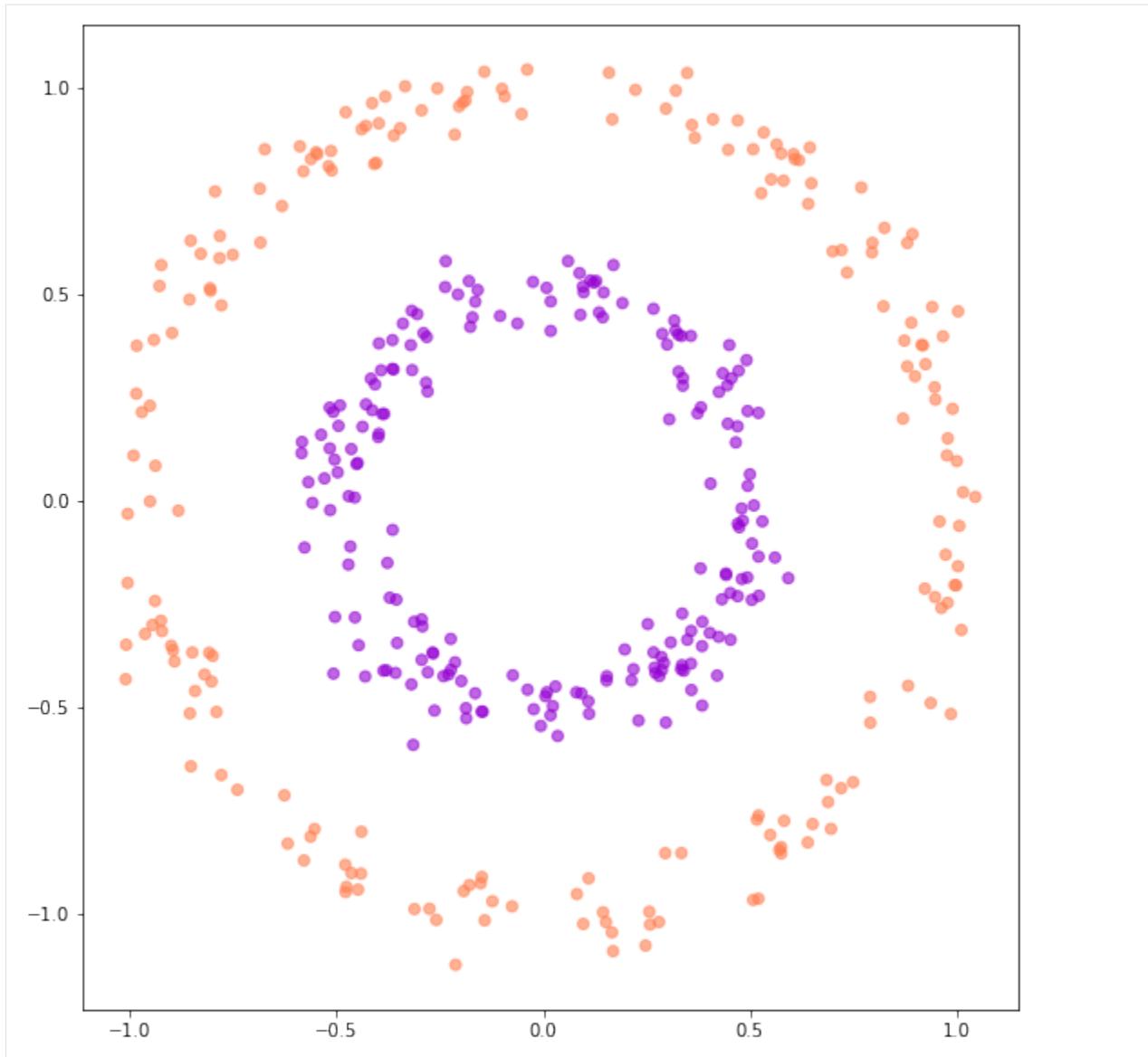
def __len__(self):
    return self.num_samples

def __getitem__(self, idx):
    return self.X[idx], self.y[idx]

class ConcentricCirclesDataModule(ToyDataModule):
    def __init__(self, dataset_size:int, split_percentages:Optional[float]=None):
        super().__init__(dataset_size, split_percentages)

    def setup(self, stage: Optional[str] = None):
        dataset = ConcentricCircles(num_samples=self.dataset_size, noise_std=5e-2)
        self.train_set, self.val_set, self.test_set = torch.utils.data.random_
        ↪split(dataset, self.split_sizes)
```

```
[ ]: sample_circles_set = ConcentricCircles(num_samples=401, noise_std=5e-2)
fig, ax = plot_binary_classification_dataset(*sample_circles_set[:])
```



We will now train a Neural ODE on this dataset. We will use a 3-layer MLP with ReLU activations and 64 hidden dimensions.

```
[ ]: circles_data_module = ConcentricCirclesDataModule(1000)

adjoint = True
t_span = torch.linspace(0, 1, 2)
f = nn.Sequential(nn.Linear(2, 64), nn.Tanh(), nn.Linear(64, 64), nn.Tanh(), nn.
    Linear(64, 2))
model = ODEBlock(f, adjoint=adjoint)
learner = Learner(model, t_span)

trainer = pl.Trainer(
    max_epochs=300,
    accelerator="gpu" if torch.cuda.is_available() else "cpu",
    devices=1,
```

(continues on next page)

(continued from previous page)

```

callbacks=[  

    pl.callbacks.ModelCheckpoint(mode="max", monitor="val_accuracy"),  

    pl.callbacks.RichProgressBar(),  

],  

log_every_n_steps=1,  

)  
  

trainer.fit(learner, datamodule=circles_data_module)
val_result = trainer.validate(learner, datamodule=circles_data_module, verbose=True)
test_result = trainer.test(learner, datamodule=circles_data_module, verbose=True)

```

GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

Name	Type	Params
0	model	ODEBlock
1	accuracy	Accuracy

Trainable params: 4.5 K
Non-trainable params: 0
Total params: 4.5 K
Total estimated model params size (MB): 0

Output()

Output()

Validate metric	DataLoader 0
val_accuracy	0.9900000095367432
val_loss	0.042514488101005554

Output()

Test metric	DataLoader 0
test_accuracy	0.9900000095367432

(continues on next page)

(continued from previous page)

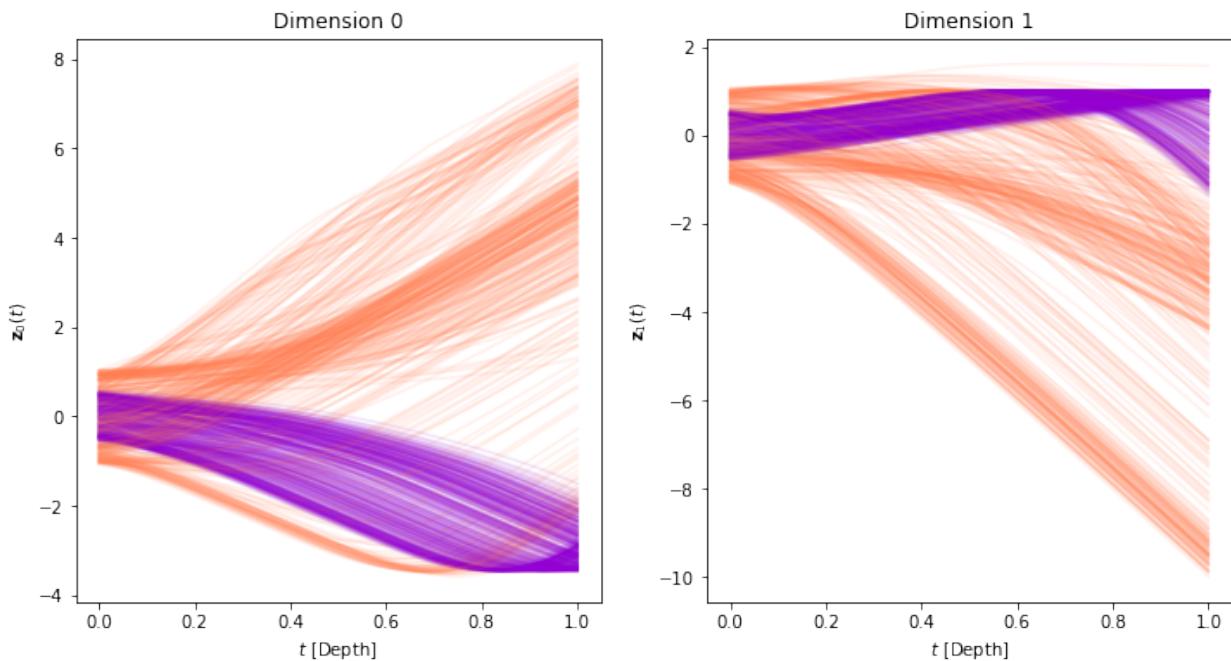
test_loss	0.037914443761110306

Interestingly, the model can achieve perfect accuracy, even though in theory it cannot separate the 2 classes. Let's visualize the trajectories and see how this can be the case. Similar to the previous experiment, we will first run inference and save the trajectories using a dense time span.

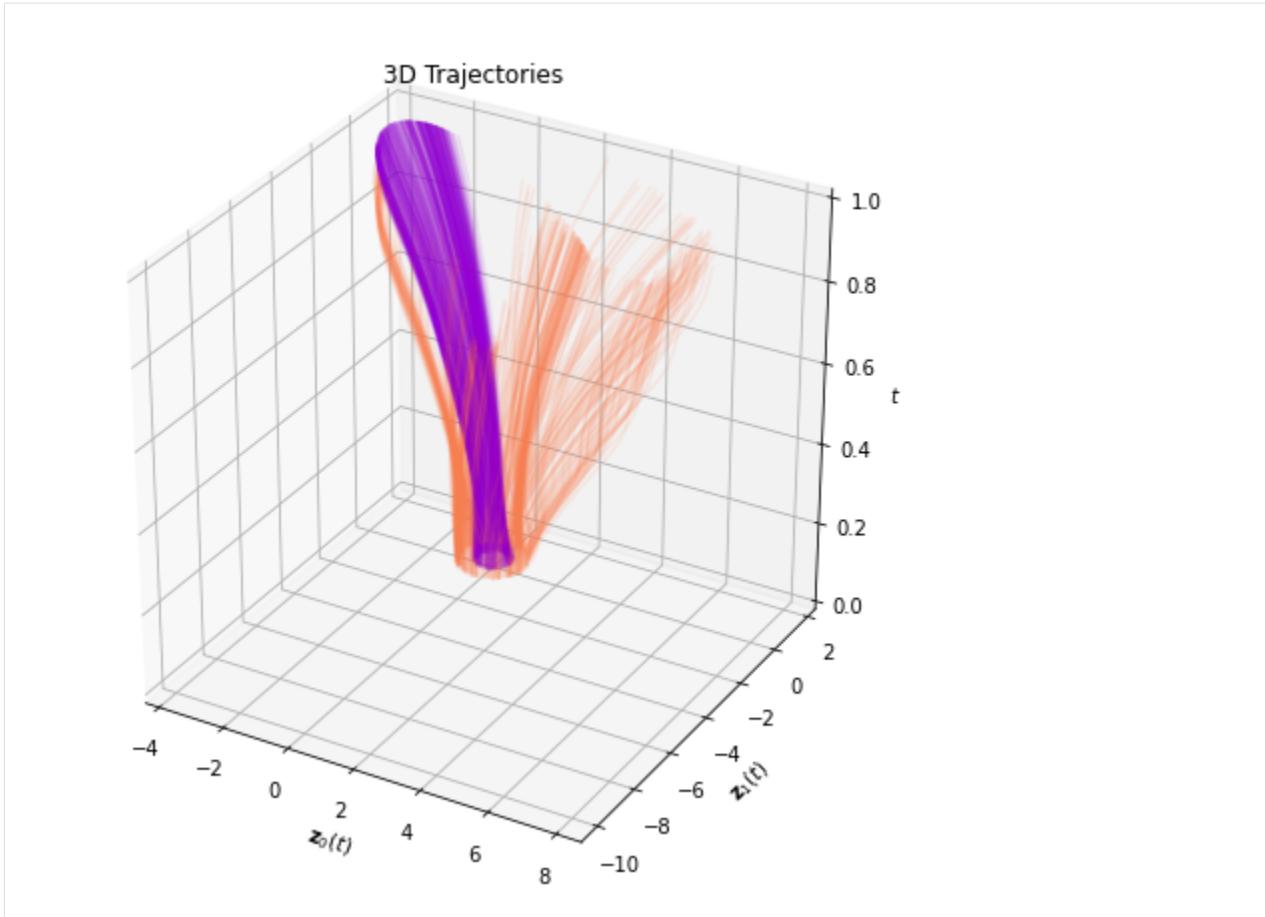
```
[ ]: num_timesteps = 100
time_span = np.linspace(0.0, 1.0, num_timesteps)
trajectories, classes = run_inference(learner, circles_data_module.train_dataloader(), time_span)

colors = ['coral', 'darkviolet']
class_colors = [colors[ci] for ci in classes]
```

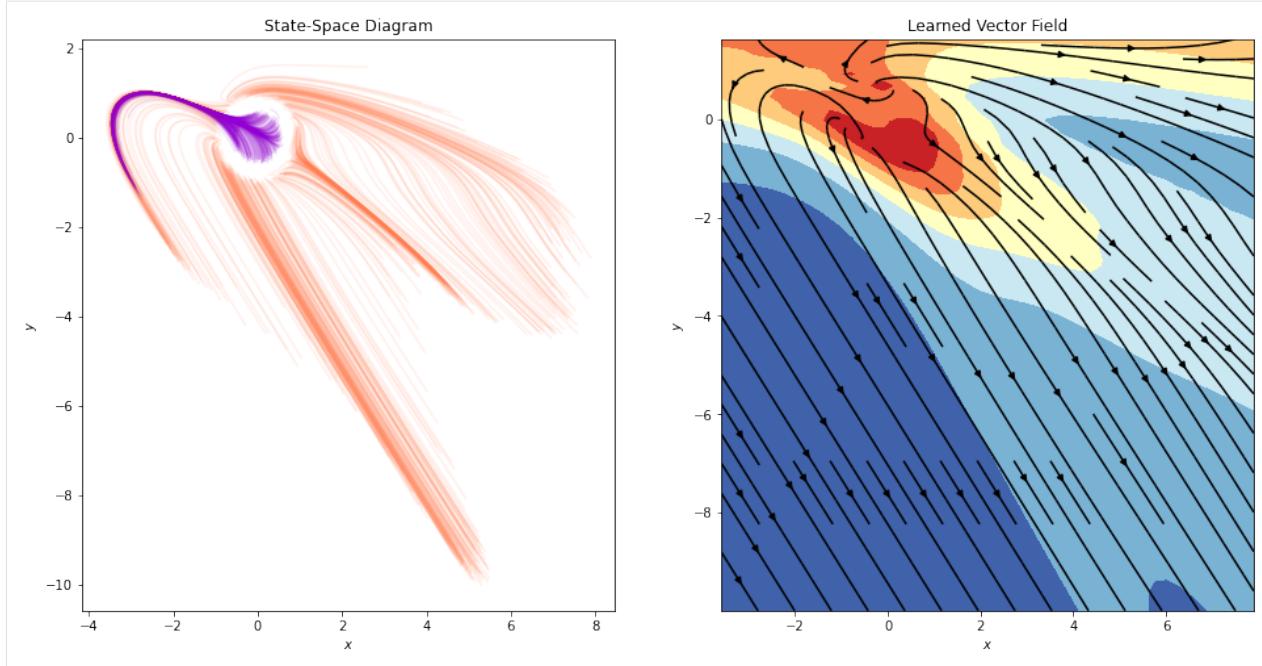
```
[ ]: plot_trajectories(time_span, trajectories, class_colors)
```



```
[ ]: plot_trajectories_3d(time_span, trajectories, class_colors)
```



```
[ ]: fig, ax = plt.subplots(1, 2, figsize=(16, 8))
plot_state_space(traj, class_colors, ax=ax[0])
plot_static_vector_field(model, traj, ax=ax[1], device=device)
```



It is not clear neither from the trajectories, nor from the state-space, how the model manages to separate the data. Let's use an animated visualization of the evolution once again.

```
[ ]: anim = plot_trajectories_animation(time_span, trajectories, colors, classes, lim=12.0)
HTML(anim.to_html5_video())
<IPython.core.display.HTML object>
```

The animation clearly shows that the model is cheating by stretching a part of space so much that data points from the inner circle can flow that region. Even though this offers perfect accuracy in this dataset, it comes at the expense of generalization capabilities, and increased train and inference time, since the network has to run many more function evaluations to achieve these results.

Augmented Neural ODEs

These issues were first observed (in the context of Neural ODEs) by Emilian Dupont et al., in their work on [**Augmented Neural ODEs**](#). The authors propose a very simple yet elegant solution to the problem: they augment the space on which they learn and solve the ODE, from \mathbb{R}^d to \mathbb{R}^{d+p} . The initial state \mathbf{x} is augmented with additional dimensions \mathbf{a} that are initialized with 0.

The augmented ODE problem is formulated as:

$$\begin{bmatrix} \dot{\mathbf{h}}(t) \\ \dot{\mathbf{a}}(t) \end{bmatrix} = \mathbf{f} \left(\begin{bmatrix} \dot{\mathbf{h}}(t) \\ \dot{\mathbf{a}}(t) \end{bmatrix} \right), \quad \begin{bmatrix} \mathbf{h}(0) \\ \mathbf{a}(0) \end{bmatrix} = \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix}$$

The final predictions in the original space are achieved via a final network that transforms the output augmented states:

$$\hat{\mathbf{y}} = \mathbf{g} \left(\begin{bmatrix} \mathbf{h}(T) \\ \mathbf{a}(T) \end{bmatrix} \right),$$

where \mathbf{g} can be an *MLP* or even a simple *linear* layer.

We will now define a simple Augmenter module, and a Neural ODE wrapper that incorporates the Augmenter and the output network.

```
[ ]: class ZeroAugmenter(nn.Module):
    def __init__(self, axis=1, num_dims=1):
        super().__init__()
        self.axis = axis
        self.num_dims = num_dims

    def forward(self, x):
        aug_dims = list(x.shape)
        aug_dims[self.axis] = self.num_dims
        augmentation = torch.zeros(aug_dims, device=x.device, dtype=x.dtype)
        return torch.cat([x, augmentation], dim=self.axis)

class AugmentedNODEWrapper(nn.Module):
    def __init__(self, augmenter, neural_ode, out_net):
        super().__init__()
        self.augmenter = augmenter
        self.neural_ode = neural_ode
        self.out_net = out_net

    def forward(self, x: torch.Tensor, adjoint: bool = True, integration_time=None):
        x = self.augmenter(x)
        x = self.neural_ode(x, adjoint, integration_time)
        x = self.out_net(x)
        return x

    def forward_no_projection(self, x: torch.Tensor, adjoint: bool = True, integration_
    ↪time=None):
        x = self.augmenter(x)
        x = self.neural_ode(x, adjoint, integration_time)
        return x
```

We will continue with the same dataset as before, namely the 2D concentric circles. We will use a single augmentation dimension and a simple linear layer $g : \mathbb{R}^3 \rightarrow \mathbb{R}^2$.

```
[ ]: circles_data_module = ConcentricCirclesDataModule(1000)

adjoint = True
t_span = torch.linspace(0, 1, 2)
augmentation_dims = 1
f = nn.Sequential(nn.Linear(2+augmentation_dims, 64), nn.Tanh(), nn.Linear(64, ↪
    ↪2+augmentation_dims))
no_augm_model = ODEBlock(f, adjoint=adjoint)
model = AugmentedNODEWrapper(ZeroAugmenter(num_dims=augmentation_dims), no_augm_model, ↪
    ↪nn.Linear(2+augmentation_dims, 2))
learner = Learner(model, t_span)

trainer = pl.Trainer(
    max_epochs=100,
    accelerator="gpu" if torch.cuda.is_available() else "cpu",
    devices=1,
    callbacks=[pl.callbacks.ModelCheckpoint(mode="max", monitor="val_accuracy")],
```

(continues on next page)

(continued from previous page)

```
    pl.callbacks.RichProgressBar(),
],
log_every_n_steps=1,
)

trainer.fit(learner, datamodule=circles_data_module)
val_result = trainer.validate(learner, datamodule=circles_data_module, verbose=True)
test_result = trainer.test(learner, datamodule=circles_data_module, verbose=True)
```

GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

Name	Type	Params
0	model	AugmentedNODEWrapper
1	accuracy	Accuracy

Trainable params: 459
Non-trainable params: 0
Total params: 459
Total estimated model params size (MB): 0

Output()

Output()

Validate metric	DataLoader 0
val_accuracy	1.0
val_loss	7.259416634042282e-06

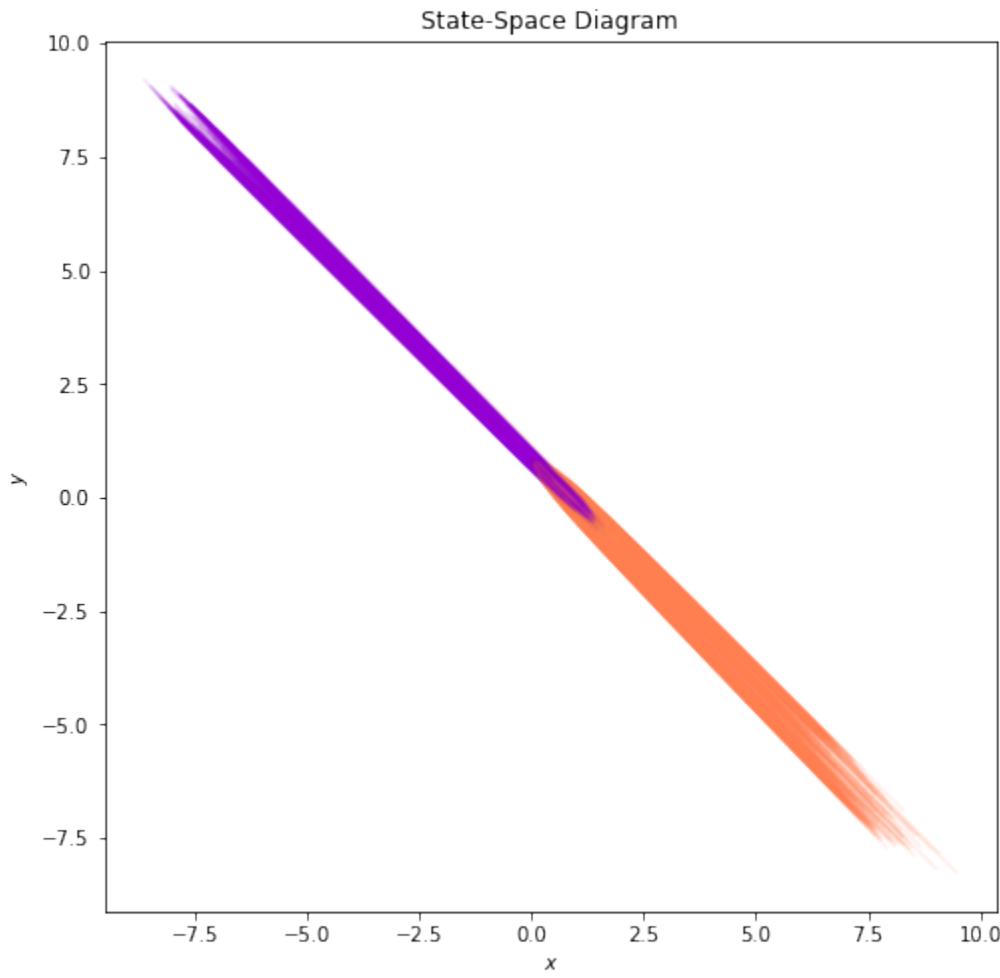
Output()

Test metric	DataLoader 0
test_accuracy	1.0
test_loss	5.69325675314758e-06

Even with a single augmentation dimension, the model can very quickly achieve perfect accuracy in the validation set. Let us now compute and visualize the trajectories, in order to gain a better understanding of what is actually going on.

```
[ ]: num_timesteps = 100
time_span = np.linspace(0.0, 1.0, num_timesteps)
trajectories, classes = run_inference(learner, circles_data_module.train_dataloader(), time_span)
```

```
[ ]: plot_state_space(trajectories, class_colors)
```



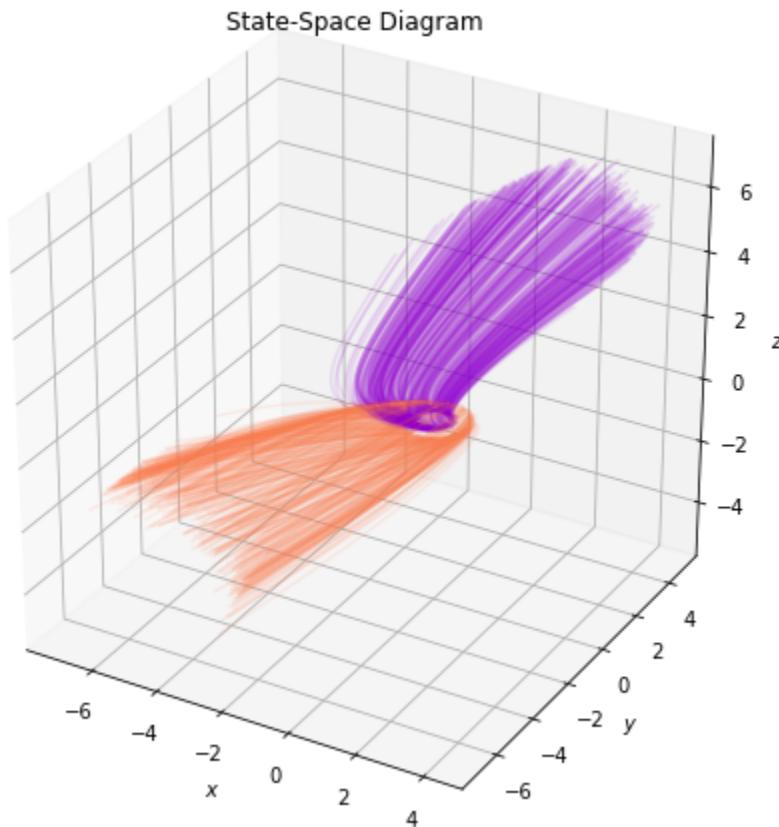
```
[ ]: anim = plot_trajectories_animation(time_span, trajectories, colors, classes, lim=8.0)
HTML(anim.to_html5_video())
<IPython.core.display.HTML object>
```

From the 2D plots, it is not exactly clear how the model can achieve perfect separation. Thus, in what follows, we will visualize the trajectories and the state-space in the augmented 3D space.

```
[ ]: @torch.no_grad()
def run_inference_no_projection(learner, data_loader, time_span):
    trajectories = []
    classes = []
    learner = learner.to(device)
    time_span = torch.from_numpy(time_span).to(device)
    for data, target in data_loader:
        data = data.to(device)
        traj = learner.inference_no_projection(data, time_span).cpu().numpy()
        trajectories.append(traj)
        classes.extend(target.numpy())
    trajectories = np.concatenate(trajectories, 1)
    return trajectories, classes

num_timesteps = 100
time_span = np.linspace(0.0, 1.0, num_timesteps)
trajectories, classes = run_inference_no_projection(learner, circles_data_module.train_
    ↵dataloader(), time_span)
colors = ['coral', 'darkviolet']
class_colors = [colors[ci] for ci in classes]
```

```
[ ]: plot_augmented_state_space(trajectories, class_colors)
```



```
[ ]: anim = plot_augmented_trajectories_animation(time_span, trajectories, colors, classes, lim=8.0)
HTML(anim.to_html5_video())
<IPython.core.display.HTML object>
```

4.41.4 Further Reading

Neural ODEs is an exciting topic that has received a lot of attention in the past few years, ever since their introduction in Neurips 2018. Some of many many work in this field include:

- Neural Stochastic Differential Equations (Neural SDEs)
- Neural Controlled Differential Equations (Neural CDEs)
- Graph ODEs
- Hamiltonian Neural Networks
- Lagrangian Neural Networks

If you want to see a quick overview of different works in the field, Michael Poli maintains the excellent [Awesome Neural ODE](#), a collection of resources regarding the interplay between neural differential equations, dynamical systems, deep learning, control, numerical methods and scientific machine learning.

[Torchdyn](#) is an excellent library for Neural Differential Equations.

[Implicit Layers](#) is a list of tutorials on implicit functions and automatic differentiation, Neural ODEs, and Deep Equilibrium Models.

[Here](#) is an excellent blogpost on ODEs and Neural ODEs.

[Patrick Kidger](#)'s doctoral dissertation is an excellent textbook on [Neural Differential Equations](#).

4.41.5 References

- [1] Chen, Ricky T. Q. et al. “**Neural ordinary differential equations**”. *NeurIPS* 2018. <https://arxiv.org/abs/1806.07366>
- [2] Dupont, Emilien et al. “**Augmented Neural ODEs**”. *NeurIPS* 2019. <https://arxiv.org/abs/1904.01681>
- [3] Pontryagin, L.S. et al. “**The mathematical theory of optimal processes**”. 1962

4.42 SGA - Sampling Discrete Structures

Notebook:

Author: Adeel Pervez

In these tutorials we discuss methods for sampling discrete variables from unstructured vectors to more structured objects such as subsets, permutations and graphs which can be incorporated in differentiable models.

At the foundation of these methods are continuous relaxations for discrete (binary or categorical) random variables. So the first part of the tutorial gives an introduction to sampling from discrete distributions with the Gumbel-Softmax trick. We use the method to train a variational autoencoder with categorical latent variables.

4.42.1 Categorical Sampling with Gumbel-Argmax

Suppose we are given a categorical distribution with C values as weights $w_i \in (0, \infty)$. We would like to obtain a sample from this distribution. The probability of each category c_i is given by the following softmax distribution

$$p_i = \frac{\exp(\log(w_i))}{\sum_j \exp(\log(w_j))}$$

The Gumbel-Argmax method for sampling this distribution is the following: Sample $U_k \sim Uniform(0, 1)$ iid and compute $r_k = \log \alpha_k - \log(-\log U_k)$. Then choose the index i of the maximum r_k (ie take the argmax) and return the 1-hot vector with the i th index set to 1 and the rest to 0. The form of noise $-\log(-\log U_k)$ added to form r_k has a Gumbel distribution whence the method gets its name. The cumulative distribution function of the Gumbel distribution (with location 0 and scale 1) is given as

$$F(z) = \exp(-\exp(-z))$$

You can take a look at a proof that this indeed samples from the softmax distribution [here](#).

In short sampling a categorical variable with the Gumbel reparameterization proceeds as follows.

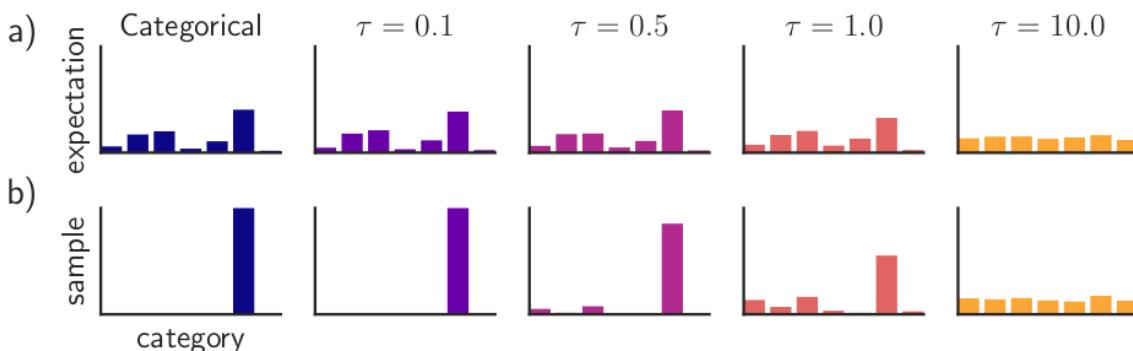
1. Given weights w_i compute $r_i = w_i + g_i$ where g_i are iid Gumbel samples
2. Argmax: Return index of largest r_i as a 1-hot vector

4.42.2 Softmax Relaxation

The above procedure still cannot be used in a differentiable model since the argmax operation has zero gradient except at points of discontinuity. So instead we use the softmax as a differentiable approximation to the argmax. In order to control the approximation we introduce a tunable temperature hyperparameter τ which controls how far the softmax outputs are from being 1-hot.

$$p_i = \frac{\exp(r_i/\tau)}{\sum_j \exp(r_j/\tau)}$$

The following figure ([Paper Link](#)) shows the effect of temperature on the distribution and samples.



A smaller temperature indicates a tighter approximation and the larger the temperature the looser the approximation. Of course, if the temperature is too small we wouldn't be able to train the model since the gradients would be very small. On the other hand, a large temperature would make the categorical outputs very far from being discrete, so it is important to choose an appropriate temperature for the problem at hand. One possibility is to slowly anneal the temperature from large to small so that close to the end of training the relaxed categorical outputs are closed to discrete. In practice, however, the temperature is often kept fixed during each training trial and tuned with cross-validation.

With the softmax relaxation the sampling then proceeds as follows

1. Given weights w_i compute $r_i = w_i + g_i$ where g_i are iid Gumbel samples
2. Apply softmax with temperature to obtain a relaxed categorical sample

4.42.3 Categorical VAE

As an example of the Gumbel Softmax relaxation we show a VAE with a categorical variable latent space for MNIST. The latent space has the structure of a vector of categorical variables each with the same fixed number of categories. In the following example the latent space has 30 categorical variables each of dimension 10. Since this is a VAE we also need to define a prior on the latent space which we define to be the uniform categorical distribution.

The following implementation uses code from [here](#) with minor modification.

We begin with the required imports and hyperparameter definitions.

```
[1]: import numpy as np

import torch
import torch.nn.functional as F
from torch import nn, optim
from torch.nn import functional as F
from torchvision import datasets, transforms
from torchvision.utils import save_image
from torch.distributions.one_hot_categorical import OneHotCategorical

import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

cuda=True
```

```
[2]: batch_size = 100
epochs = 10
latent_dim = 30
categorical_dim = 10
temp = 1.0
```

Gumbel Sampling

We now define the Gumbel-Softmax sampling routines. The `sample_gumbel` function samples scale 0 location 1 Gumbel variables by sampling uniform random variables in $U(0, 1)$ and computing $-\log(-\log(U(0, 1)))$. The categorical parameters are input as unnormalized log probabilities. The `gumbel_softmax_sample` function adds the Gumbel noise to the logits, applies the temperature and the softmax function. In the `gumbel_softmax` function we also add evaluation code which simply returns a sample (unrelaxed) from the categorical distribution parameterized by `logits`.

```
[3]: def sample_gumbel(shape, eps=1e-20):
    U = torch.rand(shape)
    if cuda:
        U = U.cuda()
    return -torch.log(-torch.log(U + eps) + eps)

def gumbel_softmax_sample(logits, temperature):
```

(continues on next page)

(continued from previous page)

```

y = logits + sample_gumbel(logits.size())
return F.softmax(y / temperature, dim=-1)

def gumbel_softmax(logits, temperature, evaluate=False):
    if evaluate:
        d = OneHotCategorical(logits=logits.view(-1, latent_dim, categorical_dim))
        return d.sample().view(-1, latent_dim * categorical_dim)

    y = gumbel_softmax_sample(logits, temperature)
    return y.view(-1, latent_dim * categorical_dim)

```

VAE model

Now we define the VAE model. The encoder computes the categorical probability parameters from which relaxed categorical variables can be sampled and passed into the decoder.

```

[4]: class VAE_gumbel(nn.Module):
    def __init__(self, temp):
        super(VAE_gumbel, self).__init__()

        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, latent_dim * categorical_dim)

        self.fc4 = nn.Linear(latent_dim * categorical_dim, 256)
        self.fc5 = nn.Linear(256, 512)
        self.fc6 = nn.Linear(512, 784)

        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def encode(self, x):
        h1 = self.relu(self.fc1(x))
        h2 = self.relu(self.fc2(h1))
        return self.relu(self.fc3(h2))

    def decode(self, z):
        h4 = self.relu(self.fc4(z))
        h5 = self.relu(self.fc5(h4))
        return self.sigmoid(self.fc6(h5))

    def forward(self, x, temp, evaluate=False):
        q = self.encode(x.view(-1, 784))
        q_y = q.view(q.size(0), latent_dim, categorical_dim)
        z = gumbel_softmax(q_y, temp, evaluate)
        return self.decode(z), F.softmax(q_y, dim=-1).reshape(*q.size())

```

KL Divergence

Since this is a VAE we also need to compute a KL divergence of the latent probability distribution with the uniform prior $p(x) = 1/C$ for all x where C is the total number of categories.

$$KLD(q||p) = E_q \left[\log \frac{q(x)}{p(x)} \right] \quad (4.105)$$

$$= \sum_{i=1}^C q(x_i) \log(C \cdot q(x_i)) \quad (4.106)$$

Here $q(x)$ is the latent probability distribution

Finally we compute the reconstruction loss of the input as the binary cross entropy between the reconstruction parameters and the input image and add that to the KL divergence to get the VAE loss.

```
[5]: def loss_function(recon_x, x, qy):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), size_average=False) / x.
    ↵shape[0]

    log_ratio = torch.log(qy * categorical_dim + 1e-20)
    KLD = torch.sum(qy * log_ratio, dim=-1).mean()

    return BCE + KLD
```

Next we build the model and train

```
[6]: model = VAE_gumbel(temp)
if cuda:
    model.cuda()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

```
[7]: kwargs = {'num_workers': 1, 'pin_memory': True} if cuda else {}
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data/MNIST', train=True, download=True,
                  transform=transforms.ToTensor()),
    batch_size=batch_size, shuffle=True, **kwargs)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data/MNIST', train=False, transform=transforms.ToTensor()),
    batch_size=batch_size, shuffle=True, **kwargs)

/home/apervez/anaconda3/envs/pytorch1.9_2/lib/python3.7/site-packages/torchvision/
    ↵datasets/mnist.py:498: UserWarning: The given NumPy array is not writeable, and
    ↵PyTorch does not support non-writeable tensors. This means you can write to the
    ↵underlying (supposedly non-writeable) NumPy array using the tensor. You may want to
    ↵copy the array to protect its data or make it writeable before converting it to a
    ↵tensor. This type of warning will be suppressed for the rest of this program.
    ↵(Triggered internally at  /opt/conda/conda-bld/pytorch_1623448224956/work/torch/csrc/
    ↵utils/tensor_numpy.cpp:180.)
    return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

For training we compute both the relaxed and the unrelaxed objective. The unrelaxed objective is not itself used for training. However, since that's the objective we want to improve, it's a good idea to also evaluate it so that we can observe how far the relaxed objective is from the actual objective. This also allows us to get an idea of how low or high to set the temperature so that the relaxed objective is not too far from the true objective while achieving reasonable training performance.

```
[8]: def train(epoch):
    model.train()
    train_loss = 0
    train_loss_unrelaxed = 0
    #temp = args.temp
    for batch_idx, (data, _) in enumerate(train_loader):
        if cuda:
            data = data.cuda()
        optimizer.zero_grad()
        recon_batch, qy = model(data, temp)
        loss = loss_function(recon_batch, data, qy)
        loss.backward()
        train_loss += loss.item() * len(data)
        optimizer.step()

        #Unrelaxed training objective for comparison
        recon_batch_eval, qy_eval = model(data, temp, evaluate=True)
        loss_eval = loss_function(recon_batch_eval, data, qy_eval)
        train_loss_unrelaxed += loss_eval.item() * len(data)

    print('Epoch: {} Average loss relaxed: {:.4f} Unrelaxed: {:.4f}'.format(
        epoch, train_loss / len(train_loader.dataset) , train_loss_unrelaxed / len(train_
        loader.dataset)))
```

```
[9]: def test(epoch):
    model.eval()
    test_loss = 0
    for i, (data, _) in enumerate(test_loader):
        if cuda:
            data = data.cuda()
        recon_batch, qy = model(data, temp, evaluate=True)
        test_loss += loss_function(recon_batch, data, qy).item() * len(data)
    test_loss /= len(test_loader.dataset)
    print('Eval loss: {:.4f}'.format(test_loss))
```

Finally we can run the training. You can try training with different values of the temperature to see how that affects the relaxed objective relative to the true one.

```
[10]: def run():
    for epoch in range(1, epochs + 1):
        train(epoch)
        test(epoch)
run()

/home/apervez/anaconda3/envs/pytorch1.9_2/lib/python3.7/site-packages/torch/nn/_reduction.py:42: UserWarning: size_average and reduce args will be deprecated, please use reduction='sum' instead.
  warnings.warn(warning.format(ret))
```

Epoch: 1 Average loss relaxed: 197.1857 Unrelaxed: 199.8391

Eval loss: 178.9625

Epoch: 2 Average loss relaxed: 154.2037 Unrelaxed: 163.6063

(continues on next page)

(continued from previous page)

```

Eval loss: 152.1406
Epoch: 3 Average loss relaxed: 136.7130 Unrelaxed: 148.5179
Eval loss: 144.5594
Epoch: 4 Average loss relaxed: 129.4279 Unrelaxed: 141.8772
Eval loss: 138.8380
Epoch: 5 Average loss relaxed: 124.0670 Unrelaxed: 137.2079
Eval loss: 135.5893
Epoch: 6 Average loss relaxed: 120.2097 Unrelaxed: 133.9328
Eval loss: 131.6892
Epoch: 7 Average loss relaxed: 117.3797 Unrelaxed: 131.4627
Eval loss: 130.6390
Epoch: 8 Average loss relaxed: 115.1937 Unrelaxed: 129.7081
Eval loss: 128.6939
Epoch: 9 Average loss relaxed: 113.3871 Unrelaxed: 128.1377
Eval loss: 128.0002
Epoch: 10 Average loss relaxed: 111.9536 Unrelaxed: 127.0920
Eval loss: 126.7100

```

Generating Samples

We can now generate some samples from the trained decoder. For this we sample some uniform categorical variables from the prior and pass them into the decoder.

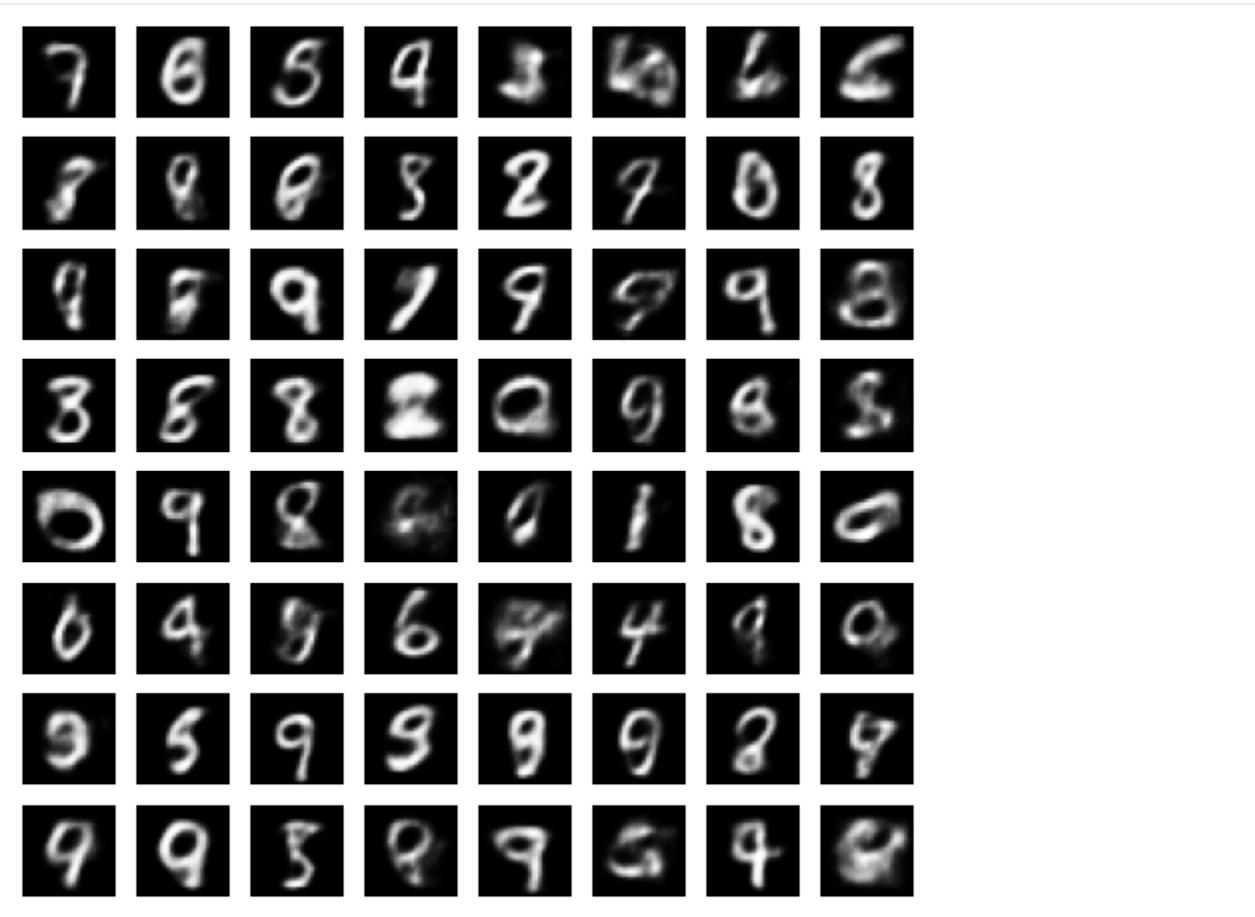
```
[11]: def generate_samples():
    #generate uniform probability vector
    model.eval()
    probs = torch.ones([64, latent_dim, categorical_dim])*(1/categorical_dim)
    cat_samples = OneHotCategorical(probs=probs.cuda()).sample().view(-1, latent_
    dim*categorical_dim)
    output = model.decode(cat_samples)
    return output.view(1, 28, 28).detach().cpu().numpy()
```

```
[12]: samples = generate_samples()
```

```
[13]: def show_gray_image_grid(imgs, x=2, y=5, size=(8,8), path=None, save=False):
    fig, axs = plt.subplots(x, y, figsize=size)
    axs = axs.flatten()
    for img, ax in zip(imgs, axs):
        ax.imshow(np.squeeze(img), cmap='gray')
        #ax.imshow(img, cmap='gray')
        ax.set_axis_off()

    if save:
        plt.savefig(path)
    else:
        plt.show()

show_gray_image_grid(samples, 8,8)
```



4.42.4 Gumbel Straight-Through

In some instances we want to train models with discrete variables but do not want to pass relaxed values as inputs. This might be the case where we want to optimize a function that cannot be defined for relaxed inputs and must use categorical/discrete inputs. One heuristic for such cases is the Straight-Through estimator. Here given some pre-activation y for which we want the gradient we compute the sample z using the non-differentiable sampling operations such as with categorical or Bernoulli sampling. Then we compute the downstream function f on the hard sample. Then in the backward pass we ignore the non-differentiable operation and pass the gradient relative to z back as the gradient relative to y . That is we set.

$$\partial_y f := \partial_z f.$$

This is the straight-through gradient and gives a biased estimate of the gradient, but in many cases this can often allow us to train models with discrete variables.

Here we use the straight-through gradient with Gumbel-Softmax relaxation. In this case we compute the Gumbel-Softmax relaxation as before. From the relaxation we compute a 1-hot vector where the index with the largest value is set to 1 and all others are set to 0. This is used as the discrete value in the downstream network and we use the straight-through gradient in the backward pass.

Given a hard vector y_{hard} and a soft vector y there is a well-known trick that can be used to incorporate straight-through gradients. We compute

```
y = (y_hard - y).detach() + y
```

which simply uses the `y_hard` in the forward pass but the gradient relative to `y` in the backward pass.

```
[14]: def gumbel_softmax(logits, temperature, evaluate=False, hard=False):
    if evaluate:
        d = OneHotCategorical(logits=logits.view(-1, latent_dim, categorical_dim))
        return d.sample().view(-1, latent_dim * categorical_dim)

    y = gumbel_softmax_sample(logits, temperature)

    if hard:
        #Straight-through gradient
        #takes the index of the largest and insert a 1.
        #all others are set to 0 obtaining a 1-hot vector.
        shape = logits.size()
        _, k = y.max(-1)

        y_hard = torch.zeros_like(logits)
        y_hard = y_hard.zero_().scatter_(-1, k.view(shape[:-1] + (1,)), 1.0)

        #This a trick to use the 1-hot value in the forward pass and the
        #relaxed gradient in the backward pass
        y = (y_hard - y).detach() + y

    return y.view(-1, latent_dim * categorical_dim)
```

As an exercise you can train the VAE model given above using Gumbel-Straight-Through by using the above function in the model definition and setting `hard=True`.

In the next tutorials we will see how the basic Gumbel relaxation method can be used to differentiably sample other discrete structures such as subsets, permutations and graphs.

4.42.5 References

[Categorical Reparameterization with Gumbel-Softmax](#)

[The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables](#)

4.43 SGA - Sampling Subsets with Gumbel-Top k Relaxations

Notebook:

Author: Adeel Pervez

In this part we show how to include a subset sampling component in differentiable models using Gumbel Top k relaxations. First we show how to build a differentiable subset sampler and then we show one application to differentiable k nearest neighbor classification.

Formally speaking we are given N elements with weights w_i . We would like to sample k elements from N without replacement. Stated otherwise, we want a k -element subset $S = \{w_{i_1}, w_{i_2}, \dots, w_{i_k}\}$ from N elements.

Given total weight $Z = \sum w_i$, the first element is sampled with probability $\frac{w_{i_1}}{Z}$, the second with probability $\frac{w_{i_2}}{Z-w_{i_1}}$ and so on for k elements. Multiplying the factors gives the following distribution for k element subsets.

$$p(S) = \frac{w_{i_1}}{Z} \frac{w_{i_2}}{Z - w_{i_1}} \cdots \frac{w_{i_k}}{Z - \sum_{j=1}^{k-1} w_{i_j}}.$$

In the introduction we showed how sampling from a categorical distribution could be recast as choosing the argmax of a set of Gumbel random variables. Relaxing the argmax with a softmax allowed us to approximate sampling from the target categorical distribution. A temperature could be used to control the extent of relaxation. In this case the the categorical probabilities are given by the softmax distrbution

$$p_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} = \frac{w_i}{\sum_j w_j}$$

It turns out that by selecting the k largest Gumbel random variables instead of just the largest we can sample subsets according to the sampling without replacement probability given above. This procedure is closely related to a procedure known by the name of weighted reservoir sampling.

Seen this way, the Gumbel-Argmax trick is a method for sampling subsets of size $k = 1$ with probabilities given by p_i . Replacing the argmax by a Top- k procedure for selecting the k largest elements generalizes the Gumbel-Argmax to sample size k subsets with probablity $p(S)$. In this case we think of the Top- k procedure as returning a k -hot vector y where $y_i = 1$ if the i th element is selected and $y_i = 0$ otherwise. Thus we represent subsets as k -hot vectors which also generalizes the representation of categorical samples as 1-hot vectors.

The unrelaxed subset sampling procedure can then be written as follows given non-negative weights w_i .

1. Compute keys $\hat{r}_i = -\log(-\log(u_i)) + \log(w_i)$ for all i and $u_i \in U(0, 1)$.
2. Return k largest keys \hat{r}_i .

4.43.1 Top k Relaxation

We can construct an unrelaxed Top k by iteratively applying the softmax k times and sampling a 1-hot categorical sample at each step. The k 1-hot categorical samples are then combined into a single k -vector. When the categorical sample gives a particular element, the log probablity for that element is set to $-\infty$ for the future iterations so that the element is never chosen again. We can relax this procedure by replacing samples from the softmax by the probabilties computed by softmax. When the softmax temperature is set to be small, the sampled and the relaxed outputs are close.

In more detail the procedure is as follows.

Unrelaxed Version

For $i = 1 \dots n$ and $j = 1 \dots k$, set $\alpha^{i+1}_{-i} := \text{softmax}(\alpha^i_{-i})$ and $\alpha^{j+1}_i = \alpha^j_i + \log(1 - a^j_i)$

Here a^j_i is a sample the categorical distribution with probabilities $p(a^j_i = 1) = \frac{\exp(\alpha^j_i / \tau)}{\sum_k \exp(\alpha^j_k / \tau)}$ and τ is a temperature.

Note that when a^j_i is a 1-hot categorical sample the $\log(1 - a^j_i)$ term in the first equation above sets the next α^{j+1}_i to $-\infty$ if $a^j_i = 1$ and leaves it unchanged otherwise. This ensures that the i th element once sampled is not sampled in the next steps. Finally we add all the k vectors as $\sum_j a^j$ and return the output as the sample.

Relaxed Version

To relax the above procedure we can replace the categorical sample at step by its expectation. In this case the update becomes

For $i = 1 \dots n$ and $j = 1 \dots k$, set $\alpha^{i+1}_i = \hat{r}_{i,j}$ and $\alpha_i^{j+1} = \alpha_i^j + \log(1 - p(a_i^j = 1))$

where $p(a_i^j = 1) = \frac{\exp(\alpha_i^j / \tau)}{\sum_k \exp(\alpha_k^j / \tau)}$ as above. At low values of τ the softmax distribution becomes close to deterministic outputs a value that is close to k -hot. The temperature variable is a hyperparameter and ideally should be annealed from larger to smaller values during the course of training. However, in most applications the temperature is left fixed per trial and tuned using cross validation. Proper tuning of temperature can have a significant effect on the performance of the model.

In the following we use code from [here]

```
[1]: import os

import torch
import torch.nn as nn
import torch.nn.functional as F

from torchvision import datasets, transforms
from torch.utils.data import Dataset, DataLoader, TensorDataset, ConcatDataset, random_
split
from torch.utils.data.sampler import SubsetRandomSampler

import random
import time
from pathlib import Path

import numpy as np

import matplotlib
import matplotlib.pyplot as plt

%matplotlib inline
```

```
[2]: gpu = torch.device('cuda')
```

4.43.2 Subset Sampler Class

The following `SubsetOperator` class implements the relaxed subset sampling procedure described above. As described the `forward` method takes a list of scores (unnormalized log probs) of some fixed dimension. We add Gumbel noise with location 0 and scale 1 and divide by the temperature. Next we apply the Top- k relaxation and return the resulting k -hot vector as the sampled subset.

```
[3]: EPSILON = np.finfo(np.float32).tiny

class SubsetOperator(torch.nn.Module):
    def __init__(self, k, tau=1.0, hard=False):
        super(SubsetOperator, self).__init__()
        self.k = k
        self.hard = hard
```

(continues on next page)

(continued from previous page)

```

self.tau = tau

def forward(self, scores):
    m = torch.distributions.gumbel.Gumbel(torch.zeros_like(scores), torch.ones_
    like(scores))
    g = m.sample()
    scores = scores + g

    # continuous top k
    khot = torch.zeros_like(scores)
    onehot_approx = torch.zeros_like(scores)
    for i in range(self.k):
        khot_mask = torch.max(1.0 - onehot_approx, torch.tensor([EPSILON]).cuda())
        scores = scores + torch.log(khot_mask)
        onehot_approx = torch.nn.functional.softmax(scores / self.tau, dim=1)
        khot = khot + onehot_approx

    if self.hard:
        # straight through
        khot_hard = torch.zeros_like(khot)
        val, ind = torch.topk(khot, self.k, dim=1)
        khot_hard = khot_hard.scatter_(1, ind, 1)
        res = khot_hard - khot.detach() + khot
    else:
        res = khot

    return res

```

You can try the sampler on some example input and various temperatures. Note that the sum of the vectors elements is always k . At lower temperatures the output should be close to k -hot.

```
[4]: sampler = SubsetOperator(k=2, tau=1.0)

x = torch.tensor([[1., 2., 3., 4.]]).to('cuda')
y = sampler(x)
print(y, y.sum())

tensor([[0.0236, 0.5672, 0.4200, 0.9892]], device='cuda:0') tensor(2., device='cuda:0')
```

4.43.3 Empirical Sampling Distribution

We empirically confirm that the k -hot relaxation generates subsets with the same distribution as the sampling without replacement distribution. For this we define a set with weights in $[1, 2, 3, 4]$ and generate 10000 subsets of size 2 using the true distribution (here with Gumbel Top k). Then we generate subsets using the relaxation given above with a fixed temperature. The samples are plotted side-by-side in a histogram.

```
[5]: w = torch.tensor([[1., 2., 3., 4.]]).to('cuda')
w = w.tile((10000, 1))
w_scores = torch.log(w)
```

Use Gumbel-Top- k to get true distribution samples.

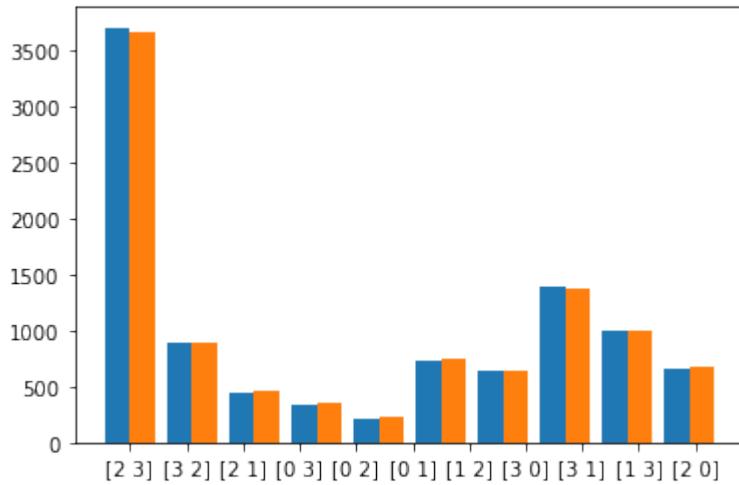
```
[6]: #true samples
m = torch.distributions.gumbel.Gumbel(torch.zeros_like(w_scores), torch.ones_like(w_
(scores))
g = m.sample()
scores = w_scores + g
samples = torch.topk(scores, 2)[1]
samples = samples.detach().cpu().numpy()
samples = [str(x) for x in samples]
```

To get samples of subsets from the relaxation, we first apply the relaxation and choose the Top k indices as the chosen subset.

```
[7]: #relaxed samples
r_samples = sampler(w_scores)

r_samples = torch.topk(r_samples, 2)[1]
r_samples = r_samples.detach().cpu().numpy()
r_samples = [str(x) for x in r_samples]
```

```
[8]: plt.hist([samples,r_samples], align='left')
plt.show()
```



4.43.4 Application: Differentiable k Nearest Neighbors

Now we apply the subset sampling procedure to a classification problem with differentiable k nearest neighbors. Recall that in k nearest neighbors classification we look at the labels of the k nearest neighbors and take the majority vote for classification. Unlike the classical form of nearest neighbors, we want to take the feature from a deep network. The k nearest neighbors loss is cannot be directly used in differentiable models so we relax it with our subset relaxation. Furthermore instead of looking for the nearest neighbors in the entire dataset (which can be large) we choose a random subset of data points for the distance calculations.

Given a query vector q and a set N of neighbors we compute the Euclidean distance between the q and each element $i \in N$. This gives us a list of scores (negative of the distances) and sample a k size subset of these scores as a relaxed k -hot vector.

Since this is a classification problem, during training we have the label y for the query vector q and for each of the neighbors y_i . If the labels are equal for a query, neighbor pair, we include the corresponding score otherwise we set it

to 0 and take the sum to compute the loss.

Given a subset of k neighbors the loss can be written as

$$L(S; q) = - \sum_{j \in S} I[y == y_i].$$

The actual loss is then the expectation of this expression over all subsets.

$$L(q) = E_S[L(S; q)].$$

The SubsetsDKNN class computes the scores between the queries and neighbors and returns the sampled subsets.

```
[9]: class SubsetsDKNN(torch.nn.Module):
    def __init__(self, k, tau=1.0, hard=False, num_samples=-1):
        super(SubsetsDKNN, self).__init__()
        self.k = k
        self.subset_sample = SubsetOperator(k=k, tau=tau, hard=hard)
        self.num_samples = num_samples

    # query: batch_size x p
    # neighbors: 10k x p
    def forward(self, query, neighbors, tau=1.0):
        diff = (query.unsqueeze(1) - neighbors.unsqueeze(0))
        squared_diffs = diff ** 2
        l2_norms = squared_diffs.sum(2)
        norms = l2_norms # .sqrt() # M x 10k
        scores = -norms

        top_k = self.subset_sample(scores)
        return top_k
```

The following is the convNet that we use to compute the features of the data examples.

```
[10]: class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, kernel_size=5, stride=1)
        self.conv2 = nn.Conv2d(20, 50, kernel_size=5, stride=1)
        self.linear = nn.Linear(800, 500)

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.conv2(out))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1)
        out = F.relu(self.linear(out))
        return out
```

Define hyperparameters.

Here we say that we are going to using 100 queries per minibatch and 100 neighbors for the distance computation. The embedding size for each data sample is set to 500.

```
[11]: k = 9
tau = 1.0
NUM_TRAIN_QUERIES = 100
NUM_TEST_QUERIES = 10
NUM_TRAIN_NEIGHBORS = 100
LEARNING_RATE = 10 ** -3
NUM_EPOCHS = 20
EMBEDDING_SIZE = 500
```

```
[12]: dknn_layer = SubsetsDKNN(k, tau)
```

Data Sampler

In theory we could use the entire dataset as the set of data points from which to choose the k nearest neighbors for any query data point. This, however, is expensive so we instead use a random set of data points from which to select the k nearest neighbors.

In the following we define a data sampler class which samples a batch of query samples and a batch of neighbors from which to select the k nearest neighbors for each query. The size of the query set and the neighbor set is fixed by the NUM_TRAIN_QUERIES and the NUM_TRAIN_NEIGHBORS hyperparameters defined above.

```
[13]: class ClassicDataset(Dataset):

    def __init__(self,
                 x,
                 y,
                 transform):
        self.xy = TensorDataset(x, y)
        self.transform = transform

    def __len__(self):
        return len(self.xy)

    def __getitem__(self, idx):
        x, y = self.xy[idx]
        if self.transform:
            x = self.transform(x)

        return x, y

class DataSplit(object):
    def __init__(self, dataset):
        if dataset == 'mnist':
            train_real = datasets.MNIST(root='./data-mnist', download=True)
```

(continues on next page)

(continued from previous page)

```

tr_real_ds, va_real_ds = random_split(trva_real, [55000, 5000])
xtr_real = trva_real.train_data[tr_real_ds.indices].view(
    -1, 1, 28, 28)
ytr_real = trva_real.train_labels[tr_real_ds.indices]
xva_real = trva_real.train_data[va_real_ds.indices].view(
    -1, 1, 28, 28)
yva_real = trva_real.train_labels[va_real_ds.indices]

trans = transforms.Compose(
    [transforms.ToPILImage(), transforms.ToTensor()])
)

self.train_dataset = ClassicDataset(
    x=xtr_real, y=ytr_real, transform=trans)
self.valid_dataset = ClassicDataset(
    x=xva_real, y=yva_real, transform=trans)
self.test_dataset = datasets.MNIST(root='./data-mnist', train=False,
transform=transforms.Compose([
    transforms.ToTensor()
]))

def get_train_loader(self, batch_size, **kwargs):
    train_loader = DataLoader(self.train_dataset,
        batch_size=batch_size, num_workers=4, shuffle=True,
**kwargs)
    return train_loader

def get_valid_loader(self, batch_size, **kwargs):
    valid_loader = DataLoader(self.valid_dataset,
        batch_size=batch_size, shuffle=True, **kwargs)
    return valid_loader

def get_test_loader(self, batch_size, **kwargs):
    test_loader = DataLoader(self.test_dataset,
        batch_size=batch_size, shuffle=False, **kwargs)
    return test_loader

```

Loss

Now we compute the loss given the sampled subsets for the queries using the labels for the queries and neighbors.

```
[14]: def dknn_loss(query, neighbors, query_label, neighbor_labels):
    # query: batch_size x p
    # neighbors: 10k x p
    # query_labels: batch_size x [10] one-hot
    # neighbor_labels: n x [10] one-hot

    # num_samples x batch_size x n
    start = time.time()
    top_k_ness = dknn_layer(query, neighbors)
```

(continues on next page)

(continued from previous page)

```

elapsed = time.time() - start
correct = (query_label.unsqueeze(1) *
           neighbor_labels.unsqueeze(0)).sum(-1) # batch_size x n
correct_in_top_k = (correct.unsqueeze(0) * top_k_ness).sum(-1)
loss = -correct_in_top_k

return loss, elapsed

```

[15]: h_phi = ConvNet().to(cuda)

[16]: optimizer = torch.optim.SGD(
 h_phi.parameters(), lr=LEARNING_RATE, momentum=0.9, weight_decay=5e-4)

[17]: split = DataSplit('mnist')

```

batched_query_train = split.get_train_loader(NUM_TRAIN_QUERIES)
batched_neighbor_train = split.get_train_loader(NUM_TRAIN_NEIGHBORS)

```

```

# labels is a 1-dimensional tensor
def one_hot(labels, l=10):
    n = labels.shape[0]
    labels = labels.unsqueeze(-1)
    oh = torch.zeros(n, l, device='cuda').scatter_(1, labels, 1)
    return oh

```

```

/home/apervez/anaconda3/envs/pytorch1.9_2/lib/python3.7/site-packages/torchvision/
datasets/mnist.py:498: UserWarning: The given NumPy array is not writeable, and
PyTorch does not support non-writeable tensors. This means you can write to the
underlying (supposedly non-writeable) NumPy array using the tensor. You may want to
copy the array to protect its data or make it writeable before converting it to a
tensor. This type of warning will be suppressed for the rest of this program.
(Triggered internally at  /opt/conda/conda-bld/pytorch_1623448224956/work/torch/csrc/
utils/tensor_numpy.cpp:180.)
    return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
/home/apervez/anaconda3/envs/pytorch1.9_2/lib/python3.7/site-packages/torchvision/
datasets/mnist.py:62: UserWarning: train_data has been renamed data
    warnings.warn("train_data has been renamed data")
/home/apervez/anaconda3/envs/pytorch1.9_2/lib/python3.7/site-packages/torchvision/
datasets/mnist.py:52: UserWarning: train_labels has been renamed targets
    warnings.warn("train_labels has been renamed targets")

```

Training

To train we sample a batch of queries and a batch of neighbors and compute the knn loss for each query and display the classification accuracy.

```
[18]: def train(epoch):
    timings = []
    h_phi.train()
    to_average = []
    # train
    for query, candidates in zip(batched_query_train, batched_neighbor_train):
        optimizer.zero_grad()
        cand_x, cand_y = candidates
        query_x, query_y = query

        cand_x = cand_x.to(device=gpu)
        cand_y = cand_y.to(device=gpu)
        query_x = query_x.to(device=gpu)
        query_y = query_y.to(device=gpu)

        neighbor_e = h_phi(cand_x).reshape(NUM_TRAIN_NEIGHBORS, EMBEDDING_SIZE)
        query_e = h_phi(query_x).reshape(NUM_TRAIN_QUERIES, EMBEDDING_SIZE)

        neighbor_y_oh = one_hot(cand_y).reshape(NUM_TRAIN_NEIGHBORS, 10)
        query_y_oh = one_hot(query_y).reshape(NUM_TRAIN_QUERIES, 10)

        losses, timing = dknn_loss(query_e, neighbor_e, query_y_oh, neighbor_y_oh)
        timings.append(timing)
        loss = losses.mean()
        loss.backward()
        optimizer.step()
        to_average.append((-loss).item() / k)

    print('Avg. train correctness of top k:',
          sum(to_average) / len(to_average))
```

For testing we can directly take the k nearest neighbors and do not sample.

```
[19]: def majority(lst):
    return max(set(lst), key=lst.count)

def new_predict(query, neighbors, neighbor_labels):
    """
    query: p
    neighbors: n x p
    neighbor_labels: n (int)
    """
    diffs = (query.unsqueeze(1) - neighbors.unsqueeze(0)) # M x n x p
    squared_diffs = diffs ** 2
    norms = squared_diffs.sum(-1) # M x n
    indices = torch.argsort(norms, dim=-1)
    labels = neighbor_labels.take(indices[:, :k]) # M x k
```

(continues on next page)

(continued from previous page)

```

prediction = [majority(l.tolist()) for l in labels]
return torch.Tensor(prediction).to(device=gpu).long()

def acc(query, neighbors, query_label, neighbor_labels):
    prediction = new_predict(query, neighbors, neighbor_labels)
    return (prediction == query_label).float().cpu().numpy()

```

[20]:

```

batched_query_val = split.get_valid_loader(NUM_TEST_QUERIES)
batched_query_test = split.get_test_loader(NUM_TEST_QUERIES)

def test(epoch, val=False):
    h_phi.eval()
    global best_acc
    with torch.no_grad():
        embeddings = []
        labels = []
        for neighbor_x, neighbor_y in batched_neighbor_train:
            neighbor_x = neighbor_x.to(device=gpu)
            neighbor_y = neighbor_y.to(device=gpu)
            embeddings.append(h_phi(neighbor_x))
            labels.append(neighbor_y)
        neighbors_e = torch.stack(embeddings).reshape(-1, EMBEDDING_SIZE)
        labels = torch.stack(labels).reshape(-1)

        results = []
        for queries in batched_query_val if val else batched_query_test:
            query_x, query_y = queries
            query_x = query_x.to(device=gpu)
            query_y = query_y.to(device=gpu)
            query_e = h_phi(query_x) # batch_size x embedding_size
            results.append(acc(query_e, neighbors_e, query_y, labels))
        total_acc = np.mean(np.array(results))

    split = 'val' if val else 'test'
    print('Avg. %s acc:' % split, total_acc)

```

We can now train the model. After 20 epochs of training we can get about 99.3% test accuracy.

[21]:

```

for t in range(1, NUM_EPOCHS):
    print('Beginning epoch %d: ' % t)
    train(t)
test(-1, val=False)

Beginning epoch 1:

```

```
/home/apervez/anaconda3/envs/pytorch1.9_2/lib/python3.7/site-packages/torch/nn/
functional.py:718: UserWarning: Named tensors and all their associated APIs are an
experimental feature and subject to change. Please do not use them for anything
important until they are released as stable. (Triggered internally at /opt/conda/
conda-bld/pytorch_1623448224956/work/c10/core/TensorImpl.h:1156.)
return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
```

```
Avg. train correctness of top k: 0.798162693676322
Beginning epoch 2:
Avg. train correctness of top k: 0.9430294330673984
Beginning epoch 3:
Avg. train correctness of top k: 0.9600246477608737
Beginning epoch 4:
Avg. train correctness of top k: 0.9682177281620528
Beginning epoch 5:
Avg. train correctness of top k: 0.9722120181960283
Beginning epoch 6:
Avg. train correctness of top k: 0.9760807987174599
Beginning epoch 7:
Avg. train correctness of top k: 0.9786814417020248
Beginning epoch 8:
Avg. train correctness of top k: 0.9803707928127704
Beginning epoch 9:
Avg. train correctness of top k: 0.9808286046500168
Beginning epoch 10:
Avg. train correctness of top k: 0.9832377206436321
Beginning epoch 11:
Avg. train correctness of top k: 0.9841142041755441
Beginning epoch 12:
Avg. train correctness of top k: 0.9854074607232604
Beginning epoch 13:
Avg. train correctness of top k: 0.9857420248937129
Beginning epoch 14:
Avg. train correctness of top k: 0.9867219129234854
Beginning epoch 15:
Avg. train correctness of top k: 0.987558315161503
Beginning epoch 16:
Avg. train correctness of top k: 0.9883937643031893
Beginning epoch 17:
Avg. train correctness of top k: 0.988576520091355
Beginning epoch 18:
Avg. train correctness of top k: 0.9895413443054827
Beginning epoch 19:
Avg. train correctness of top k: 0.98968997531467
Avg. test acc: 0.9918
```

4.43.5 References

Reparameterizable Subset Sampling via Continuous Relaxations. [Code]

4.44 SGA: Learning Latent Permutations with Gumbel-Sinkhorn Networks

Notebook:

Author: David Knigge

Code in this tutorial was based on: <https://github.com/perrying/gumbel-sinkhorn> and https://github.com/google/gumbel_sinkhorn.

4.44.1 0. Introduction

In this notebook, we take a look at implementing a flavour of neural networks that can perform operations on discrete objects. More specifically, we look at **latent permutation**, where the objective is to find the correct **permutation** P of a latent variable without direct supervision. We make use of the **Gumbel-Sinkhorn operator** which allows us to pose the finding of a permutation P as a linear optimization problem. We recommend taking a look at the original paper by Mena, G., et al., (2018), though we will briefly introduce all necessary concepts in this notebook.

Questions and feedback may be forwarded to David Knigge; d.m.knigge@uva.nl.

0.1 Brief overview of the problem

Typically, we think of the characteristic of neural networks being flexible enough to learn arbitrarily complex mappings as favourable. However, in many practical applications of neural networks, we can restrict the space of learnable functions by incorporating **inductive biases**; things we a priori know to be true (and relevant) in our problem setting.

An example of the importance of incorporating such inductive biases in model design is in settings where we are performing operations on discrete objects. Think for example of sorting a list of numbers \mathbf{n} :

$$\mathbf{n}^T = \begin{bmatrix} 5 \\ 1 \\ 3 \\ 4 \end{bmatrix}$$

We could obtain a reordering of the numbers by left-multiplying with a permutation matrix P :

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Applying such a permutation matrix to our input, we get:

$$P\mathbf{n}^T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 5 \\ 1 \\ 3 \\ 4 \end{bmatrix} \quad (4.107)$$

$$= \begin{bmatrix} 1 \\ 3 \\ 4 \\ 5 \end{bmatrix} \quad (4.108)$$

A natural first instinct would be to let our network learn the permutation matrix P directly. But what if we don't have the ground truth permutation matrix? We may instead optimize our network for the permuted input $P\mathbf{n}^T$.

However, next we should note that the matrix P has some special properties. It is what's known as an instance of a **doubly stochastic matrix**, essentially meaning both its rows and its columns sum to a value of 1. Each of its rows and columns are one-hot vectors, reflecting the fact that each value in the input \mathbf{n}^T must occur once in the output $P\mathbf{n}^T$, and only at a single location.

How do we get our network to learn to map to such doubly stochastic matrices? This is exactly the focus of the Gumbel-Sinkhorn operator!

Note: - A permutation matrix is an instance of an orthogonal matrix. Remember that for such matrices, their inverses are given by their transpose. Hence $P^T P = \mathbf{I}$. We use this later on.

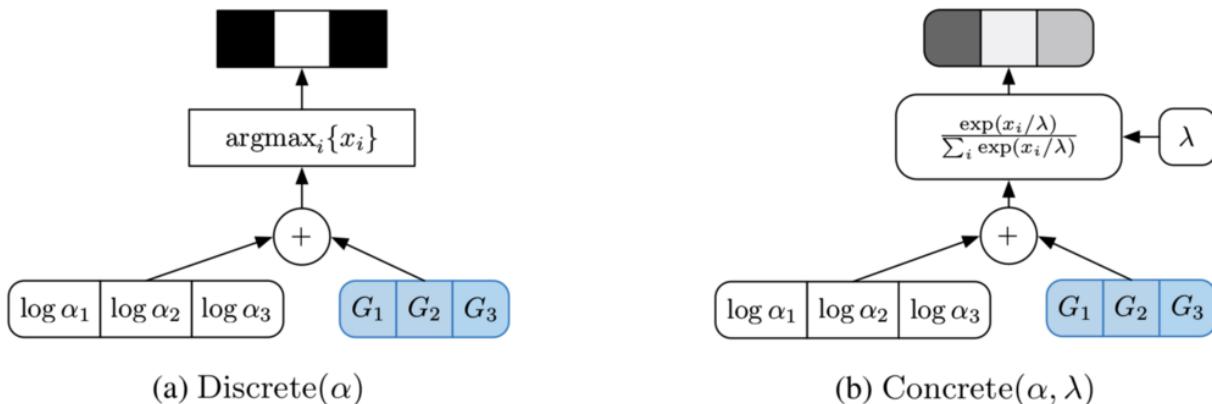
0.2 Refresher: The Gumbel-Softmax distribution

The authors introduce the Sinkhorn operator as permutation-friendly analogy to the **Gumbel-Softmax/Concrete** distribution; a differentiable approximation to sampling discrete data. Let's do a quick recap of the Gumbel-Softmax distribution.

Where the reparameterization trick as used in VAE's allows us to differentiably sample from continuous distributions, it is not straightforward to apply this to discrete distributions.

We can sample from a categorical distribution with class probabilities $\alpha = [\alpha_1, \alpha_2, \alpha_3]$ using the **Gumbel-Max trick**, by taking the argmax of $\log \alpha + \mathbf{G}$, where $\mathbf{G} = [G_1, G_2, G_3] \sim \text{Gumbel}$. However, since the argmax operator isn't differentiable, we can't get gradients from this operation (visualized below in figure a).

Instead, we **relax** the discrete distribution by approximating a sampling from it through the Gumbel-Softmax trick. We replace the argmax operation by a softmax with temperature parameter λ . This operation is differentiable!



Where $\mathbf{G}_{-i} \sim \text{Gumbel} = -\log(-\log(rmUnif(0, 1)))$.

0.3 Installing and importing some useful packages

Here we install and import some libraries that we will use throughout this tutorial. We use the pytorch as our deep learning framework of choice.

```
[1]: ## Standard libraries
import os
import numpy as np

## Imports for plotting
import matplotlib.pyplot as plt
import matplotlib
```

(continues on next page)

(continued from previous page)

```
## PyTorch
import torch
import torch.nn as nn
import torch.utils.data as data
import torch.optim as optim
## Torchvision
import torchvision
from torchvision.datasets import MNIST
from torchvision import transforms
```

[2]: # Download a pretrained model.

```
if not os.path.exists('saved_models'):
    os.mkdir('saved_models')
!wget -O 'saved_models/model.pth' https://raw.githubusercontent.com/david-knigge/uvalc2-tutorial-Gumbel-Sinkhorn-Networks/main/model.pth

--2022-04-22 12:23:42-- https://raw.githubusercontent.com/david-knigge/uvalc2-tutorial-Gumbel-Sinkhorn-Networks/main/model.pth
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 31375 (31K) [application/octet-stream]
Saving to: 'saved_models/model.pth'

100%[=====] 31,375      --.-K/s   in 0.001s

2022-04-22 12:23:42 (39.1 MB/s) - 'saved_models/model.pth' saved [31375/31375]
```

[3]: # Path to the folder where the datasets are be downloaded (e.g. MNIST)

```
DATASET_PATH = "./data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "./saved_models/"

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
```

4.44.2 1. The Gumbel-Sinkhorn operator

1.1 The Gumbel-Sinkhorn operator

In 0.2, we've seen how we can approximate a discrete distribution with continuous values by a temperature-dependent softmax function;

$$\text{softmax}_\tau(x_i) = \frac{\exp \frac{x_i}{\tau}}{\sum_j \exp \frac{x_j}{\tau}}$$

If we let the temperature parameter τ in this softmax approach 0, in the limit we obtain a one-hot vector corresponding to the largest x_i . We've found a differentiable way of approaching a categorical distribution! Extending this approximation to use for permutations is the point of the Sinkhorn operator.

The authors define the Sinkhorn operator S as follows:

$$S^0(X) = \exp(X) \tag{4.109}$$

$$S^l(X) = \mathcal{T}_c(\mathcal{T}_r(S^{l-1}(X))) \tag{4.110}$$

$$S(X) = \lim_{l \rightarrow \infty} S^l(X) \tag{4.111}$$

Where $\mathcal{T}_c(X) = X \oslash \mathbf{1}_N \mathbf{1}_N^T X$ and $\mathcal{T}_r(X) = X \oslash X \mathbf{1}_N \mathbf{1}_N^T$ with \oslash denoting elementwise division. Essentially, we divide each row value by the sum of its row, and each column value by the sum of its column. Check if you understand these operations. The authors note that, in the limit, $S(X)$ converges to a doubly stochastic matrix! Just the kind we need.

Let's implement the sinkhorn operator!

Notes: - For numerical stability, the Sinkhorn operator is implemented in the log-space. Remember $\log(\frac{a}{b}) = \log(a) - \log(b)$! - To ensure positivity, the effective input to sinkhorn has to be `exp(log_alpha)` (elementwise). However, note that it is only at return time that entries are exponentiated. - We make use of the `logsumexp` function to first move α out of the log domain, then sum over α (rows and columns) and then return α to the log domain.

```
[4]: def log_sinkhorn(log_alpha, n_iter):
    """Performs incomplete Sinkhorn normalization to log_alpha.
    By a theorem by Sinkhorn and Knopp [1], a sufficiently well-behaved matrix
    with positive entries can be turned into a doubly-stochastic matrix
    (i.e. its rows and columns add up to one) via the successive row and column
    normalization.

    [1] Sinkhorn, Richard and Knopp, Paul.
    Concerning nonnegative matrices and doubly stochastic
    matrices. Pacific Journal of Mathematics, 1967

    Args:
        log_alpha: 2D tensor (a matrix of shape [N, N])
            or 3D tensor (a batch of matrices of shape = [batch_size, N, N])
        n_iters: number of sinkhorn iterations (in practice, as little as 20
            iterations are needed to achieve decent convergence for N~100)
    Returns:
        A 3D tensor of close-to-doubly-stochastic matrices (2D tensors are
        converted to 3D tensors with batch_size equals to 1)
    """
    for _ in range(n_iter):
        log_alpha = log_alpha - torch.logsumexp(log_alpha, -1, keepdim=True)
        log_alpha = log_alpha - torch.logsumexp(log_alpha, -2, keepdim=True)
    return log_alpha.exp()
```

Let's see whether this function indeed yields a doubly stochastic matrix when we input the log of a matrix of unnormalized probabilities.

```
[5]: # Create a matrix containing random numbers.
X = torch.rand((3, 3))
X
```

```
[5]: tensor([[0.3834, 0.6550, 0.0086],
           [0.4759, 0.8863, 0.0279],
           [0.5294, 0.4645, 0.8727]])
```

```
[6]: # Apply the Sinkhorn operator for 20 iterations.
```

```
S_X = log_sinkhorn(torch.log(X), n_iter=20)
S_X
```

```
[6]: tensor([[0.4716, 0.4786, 0.0498],
           [0.4201, 0.4649, 0.1150],
           [0.1083, 0.0565, 0.8352]])
```

```
[7]: # Check whether rows sum to 1.
```

```
assert torch.allclose(S_X.sum(dim=0), torch.ones(S_X.shape[0]))
```

```
# Check whether columns sum to 1.
```

```
assert torch.allclose(S_X.sum(dim=1), torch.ones(S_X.shape[1]))
```

1.2 Differentiably (deterministically) obtaining a permutation matrix

1.2.1 The problem with assignment probabilities

The Sinkhorn operator we defined above is only guaranteed to give us a doubly stochastic matrix X (rows and columns sum to 1) containing assignment probabilities. It is not directly clear how to get from this doubly stochastic matrix X to an actual permutation matrix P (where rows and columns are one-hot vectors). As illustration, given a doubly stochastic matrix X :

$$X = \begin{bmatrix} 0.3 & 0.3 & 0.4 \\ 0.2 & 0.3 & 0.5 \\ 0.5 & 0.4 & 0.1 \end{bmatrix}.$$

Simply taking the arg max of every row would give us:

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix},$$

Which does not yield a valid permutation matrix.

1.2.2 From assignment probabilities a permutation matrix to as linear assignment problem

Taking a step back, the authors note that we can frame the assignment of a category v_i given a set of corresponding assignment probabilities x_i as a maximization problem. Given a one-hot vector v and a vector x , we can pick the largest x_i by choosing v such that the inner product $\langle x, v \rangle$ is maximized.

In similar fashion, the authors note that we can frame the hard choice of a permutation matrix P given a doubly stochastic square matrix X as the solution to the [linear assignment problem](#). Let \mathcal{P}_N denote the set of all possible permutation matrices P and $\langle A, B \rangle_F$ the Frobenius inner product of matrices given by $\text{trace}(A^T B)$;

$$M(X) = \arg \max_{P \in \mathcal{P}_N} \langle P, X \rangle_F$$

The authors call this operator M the matching function. Intuitively; the best hard choice of permutation matrix P given a doubly stochastic matrix X of assignment probabilities is that P whose inverse most closely maps X to the identity matrix (as this would be the matrix with the largest trace).

We implement code for solving the assignment problem below, using a [scipy implementation](#).

Notes: - Indeed, this way of finding a hard permutation matrix is non-differentiable, so it is of little use to us when training a neural net! We will only use this functionality at test time. - The `linear_sum_assignment` function expects a cost matrix as input. We instead have a probability matrix, hence we negate the probability matrix before inputting it
- We go from `row` and `col` lists of the indices of our permutation matrix we should set to 1, to a permutation matrix P using ``coo_matrix <https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html>``__.

```
[8]: ## Scipy
from scipy.optimize import linear_sum_assignment
from scipy.sparse import coo_matrix

def matching(alpha):
    # Negate the probability matrix to serve as cost matrix. This function
    # yields two lists, the row and column indices for all entries in the
    # permutation matrix we should set to 1.
    row, col = linear_sum_assignment(-alpha)

    # Create the permutation matrix.
    permutation_matrix = coo_matrix((np.ones_like(row), (row, col))).toarray()
    return torch.from_numpy(permutation_matrix)
```

Let's see whether we obtain a hard permutation matrix for the doubly stochastic matrix we created above.

```
[9]: matching(S_X)
[9]: tensor([[1, 0, 0],
           [0, 1, 0],
           [0, 0, 1]])
```

Cool! However, as mentioned above, this way of finding permutation matrices is **deterministic** and **non-differentiable** and hence will not do us much good when training a neural network to parameterize the variable α of a latent distribution.

1.3 Sampling a permutation matrix: the Gumbel-Matching distribution

The authors note that, in order to differentiably sample from a latent distribution such as the distribution of permutation matrices, we would like to apply techniques similar to the reparameterization trick that enable VAE's to learn. However, our latent distribution is discrete.

To set the stage, the authors recall the intuition behind the Gumbel-Softmax distribution; (1) with the Gumbel-max trick, we are able to sample from a categorical distribution but (2) since this reparameterization is non-differentiable (as we saw before) we instead relax the Gumbel distribution using the softmax operation, and thus obtain the Gumbel-Softmax distribution.

The authors introduce the Gumbel-Matching distribution $\mathcal{G.M.}$, the distribution of permutation matrices we get when applying i.i.d. Gumbel noise to each of the entries of an assignment probability matrix X .

$$M(X + \epsilon) \quad \mathcal{G.M.}, \text{ with } \epsilon \text{ Gumbel}$$

As in the categorical case, sampling from this distribution is still not differentiable however, as it still requires us to find the solution to a linear assignment problem.

1.3 Gumbel-Sinkhorn: Continuous relaxation of the Gumbel-Matching distribution

Now we come to the main theoretical contribution of the paper. The authors first show that the solution to the matching function $M(X)$ can in fact be obtained as the limit of the Sinkhorn operator $S(\frac{X}{\tau})$ with a small value of τ .

Finally, the authors introduce the Gumbel-Sinkhorn distribution $\mathcal{G.S.}$ as the distribution obtained by applying the Sinkhorn operator on an unnormalized assignment probability matrix to which we add i.i.d. Gumbel noise:

$$S(\frac{X + \epsilon}{\tau}) \quad \mathcal{G.S.}, \text{ with } \epsilon \text{ Gumbel.}$$

The authors show that in the limit, samples from the Gumbel-Sinkhorn distribution almost surely converge to the Gumbel-Matching distribution. The Sinkhorn operator is differentiable and may serve as a continuous relaxation for $M(X)$, allowing us to backpropagate through it!

For proof of this finding, we kindly refer you to appendix A of the paper.

Let's implement the Gumbel-Sinkhorn distribution!

```
[10]: def sample_gumbel(shape, device='cpu', eps=1e-20):
    """Samples arbitrary-shaped standard gumbel variables.
    Args:
        shape: list of integers
        eps: float, for numerical stability
    Returns:
        A sample of standard Gumbel random variables
    """
    u = torch.rand(shape, device=device)
    return -torch.log(-torch.log(u + eps) + eps)
```

Let's sample some Gumbel noise.

```
[11]: sample_gumbel((3, 3))
[11]: tensor([[ 0.1238,  0.9551, -0.6666],
              [ 1.1801, -0.8376, -0.1454],
              [-0.9623,  3.0603,  0.2785]])
```

```
[12]: def gumbel_sinkhorn(log_alpha, tau, n_iter):
    """ Sample a permutation matrix from the Gumbel-Sinkhorn distribution
    with parameters given by log_alpha and temperature tau.

    Args:
        log_alpha: Logarithm of assignment probabilities. In our case this is
            of dimensionality [num_pieces, num_pieces].
        tau: Temperature parameter, the lower the value for tau the more closely
            we follow a categorical sampling.
    """
    # Sample Gumbel noise.
    gumbel_noise = sample_gumbel(log_alpha.shape, device=log_alpha.device)

    # Apply the Sinkhorn operator!
    sampled_perm_mat = log_sinkhorn((log_alpha + gumbel_noise)/tau, n_iter)
    return sampled_perm_mat
```

Let's sample some permutation matrices.

```
[13]: # For low values of tau the sampled matrices are closer to actual
# permutation matrices.
gumbel_sinkhorn(X, tau=0.01, n_iter=20)
```

```
[13]: tensor([[1.0931e-15, 1.0000e+00, 1.6772e-38],
           [0.0000e+00, 2.3445e-10, 1.0000e+00],
           [1.0000e+00, 0.0000e+00, 0.0000e+00]])
```

```
[14]: # For higher values of tau the sampled matrices are 'less categorical'
gumbel_sinkhorn(X, tau=10, n_iter=20)
```

```
[14]: tensor([[0.3739, 0.3147, 0.3113],
           [0.3008, 0.3512, 0.3480],
           [0.3252, 0.3341, 0.3407]])
```

Importantly, this operation is differentiable!

```
[15]: # Create a matrix containing random numbers. Let PyTorch know we want gradients.
X = torch.rand((3, 3), requires_grad=True)

# Sample a permutation matrix from the Gumbel-Sinkhorn distribution.
P = gumbel_sinkhorn(X*1000, tau=1, n_iter=2000)
P.sum().backward()

X.grad
```

```
[15]: tensor([[ 0.0000e+00,  0.0000e+00,  0.0000e+00],
           [ 1.2052e-11, -1.2052e-11,  3.0366e-38],
           [ 0.0000e+00,  0.0000e+00,  0.0000e+00]])
```

4.44.3 2. Implementing a Sinkhorn Network for learning latent permutations

We now have everything in place to build our own Sinkhorn network! We will be solving the task of unscrambling MNIST digits. First, let's implement the network.

2.1 Sinkhorn Convolutional Network

We replicate the convolution-based Sinkhorn network that the authors visualize in figure 1 (see below).

```
[16]: import torch
import torch.nn as nn

class SinkhornConvNet(nn.Module):
    def __init__(self,
                 in_channels,
                 num_pieces,
                 image_size,
                 hidden_channels,
                 kernel_size,
                 tau: float = 1.0,
                 n_sink_iter: int = 20,
                 ):
        super().__init__()

        # store these for later use.
        self.tau = tau
        self.n_sink_iter = n_sink_iter

        self.g_1 = nn.Sequential(
            nn.Conv2d(in_channels, hidden_channels, kernel_size, padding=kernel_size//2),
            nn.MaxPool2d(2),
            nn.ReLU(True),
            nn.BatchNorm2d(hidden_channels)
        )

        # calculate the size of a single piece in pixels
        piece_size = image_size // num_pieces

        # calculate the size of a single piece in pixels after 1 max pooling
        piece_size_after_conv = (piece_size) // (2 * 1)

        self.g_2 = nn.Linear(piece_size_after_conv**2 * hidden_channels, num_pieces**2, ↴
                           bias=False)

    def forward(self, batch_pieces):
        batch_size = batch_pieces.shape[0]

        # Switch batch and piece dimensions. We want to apply the same network
        # to each of the pieces.
        pieces = batch_pieces.transpose(0, 1).contiguous()

        # Apply g_1 to each of the pieces.
```

(continues on next page)

(continued from previous page)

```

conv_pieces = []
for piece in pieces:
    piece = self.g_1(piece)
    conv_piece = piece.reshape(batch_size, -1)
    conv_pieces.append(conv_piece)

# Apply g_2 to each of the pieces.
latent_pieces = []
for piece in conv_pieces:
    latent_piece = self.g_2(piece)
    latent_pieces.append(latent_piece)

# Create a matrix of log unnormalized assignment probabilities. After this
# the batch dimension is batch in the first position.
log_alphas = torch.stack(latent_pieces, 1)

# During training, we sample from the Gumbel-Sinkhorn distribution.
if self.training:
    permutation_matrices = gumbel_sinkhorn(log_alphas, tau=self.tau, n_iter=self.
→n_sink_iter)

# During eval, we solve the linear assignment problem.
else:
    permutation_matrices = torch.stack([
        matching(log_alpha)
        for log_alpha in log_alphas.cpu().detach().numpy()])
    ).float().to(log_alphas.device)

# We obtain the ordered pieces as predicted by our network
ordered_pieces = inverse_permutation_for_image(batch_pieces, permutation_
→matrices)

# Return the ordered pieces, along with the predicted permutation.
# We will inspect the predicted permutation matrices during test time.
return ordered_pieces, permutation_matrices

```

2.2 Unscrambling MNIST digits

2.2.1 Scrambling MNIST.

First, we will define some utility functions for chunking and scrambling the images.

```
[17]: def chunk_image(image: torch.Tensor, num_pieces: int):
    """Randomly chunk a single image.
    Args:
        image: Image [channels, height, width].
    Returns:
        pieces: Image chunks in their original positions. [num_pieces, channels,
            height // num_pieces, width // num_pieces]
```

(continues on next page)

(continued from previous page)

```

random_pieces: Image chunks in their randomly permuted positions.
permute_index: List of permuted indices.
"""
# Get image dimensions.
height, width = image.shape[-2:]

# Get piece dimensions.
piece_height = height // num_pieces
piece_width = width // num_pieces
pieces = []

# Obtain indices for each of the image chunks.
for p_h in range(num_pieces):
    for p_w in range(num_pieces):
        left = p_w * piece_width
        right = left + piece_width
        top = p_h * piece_height
        bottom = top + piece_height
        piece = image[:,top:bottom,left:right]
        pieces.append(piece)

pieces = torch.stack(pieces, 0)

# Randomly permute the index of the pieces.
permute_index = torch.randperm(num_pieces**2)
random_pieces = pieces[permute_index]
return pieces, random_pieces, permute_index

def batch_chunk_image(images: torch.Tensor, num_pieces: int):
    """Randomly chunk a batch of images.
    Args:
        image: Images [batch, channels, height, width].
    Returns:
        pieces: Batch of image chunks in their original positions. [batch,
            num_pieces, channels, height // num_pieces, width // num_pieces]
        random_pieces: Batch of image chunks in their randomly permuted positions.
            [batch, num_pieces, channels, height // num_pieces, width // num_pieces]
        permute_index: Batch of permutation lists. [batch, num_pieces**2]
    """
    batch_pieces, batch_random_pieces, batch_permute_index = [], [], []
    for image in images:
        pieces, random_pieces, permute_index = chunk_image(image, num_pieces)

        batch_pieces.append(pieces)
        batch_random_pieces.append(random_pieces)
        batch_permute_index.append(permute_index)
    return torch.stack(batch_pieces, 0), torch.stack(batch_random_pieces, 0), torch.
    stack(batch_permute_index, 0)

def inverse_permutation_for_image(X, permutation_matrix):

```

(continues on next page)

(continued from previous page)

```
"""Apply the inverse of a permutation (its transpose) to a batch of image
chunks.

Args:
    X: Batched sets of image chunks. [batch, num_pieces, channels, height, width]
    permutation_matrix: float, for numerical stability

Returns:
    Permuted set of image chunks.
"""

return torch.einsum("bpq,bpchw->bqchw", (permutation_matrix, X)).contiguous()
```

Let's load the MNIST dataset and have a look.

```
[18]: from torchvision import datasets
from torchvision import transforms

from torch import optim
from torch.utils.data import DataLoader

device = torch.device('cuda')

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

trainset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
testset = datasets.MNIST(root='./data', train=False, transform=transform)

train_loader = DataLoader(trainset, 64, drop_last=True, shuffle=True)
test_loader = DataLoader(testset, 64, drop_last=False, shuffle=True)

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/
raw/train-images-idx3-ubyte.gz
  0%|          | 0/9912422 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/
raw/train-labels-idx1-ubyte.gz
  0%|          | 0/28881 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/
raw/t10k-images-idx3-ubyte.gz
  0%|          | 0/1648877 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw
```

(continues on next page)

(continued from previous page)

```
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/
└─raw/t10k-labels-idx1-ubyte.gz
  0%|          | 0/4542 [00:00<?, ?it/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

/home/apervez/anaconda3/envs/pytorch1.9_2/lib/python3.7/site-packages/torchvision/
└─datasets/mnist.py:498: UserWarning: The given NumPy array is not writeable, and
└─PyTorch does not support non-writeable tensors. This means you can write to the
└─underlying (supposedly non-writeable) NumPy array using the tensor. You may want to
└─copy the array to protect its data or make it writeable before converting it to a
└─tensor. This type of warning will be suppressed for the rest of this program.
└─(Triggered internally at /opt/conda/conda-bld/pytorch_1623448224956/work/torch/csrc/
└─utils/tensor_numpy.cpp:180.)
    return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

```
[19]: image_batch, _ = next(iter(test_loader))

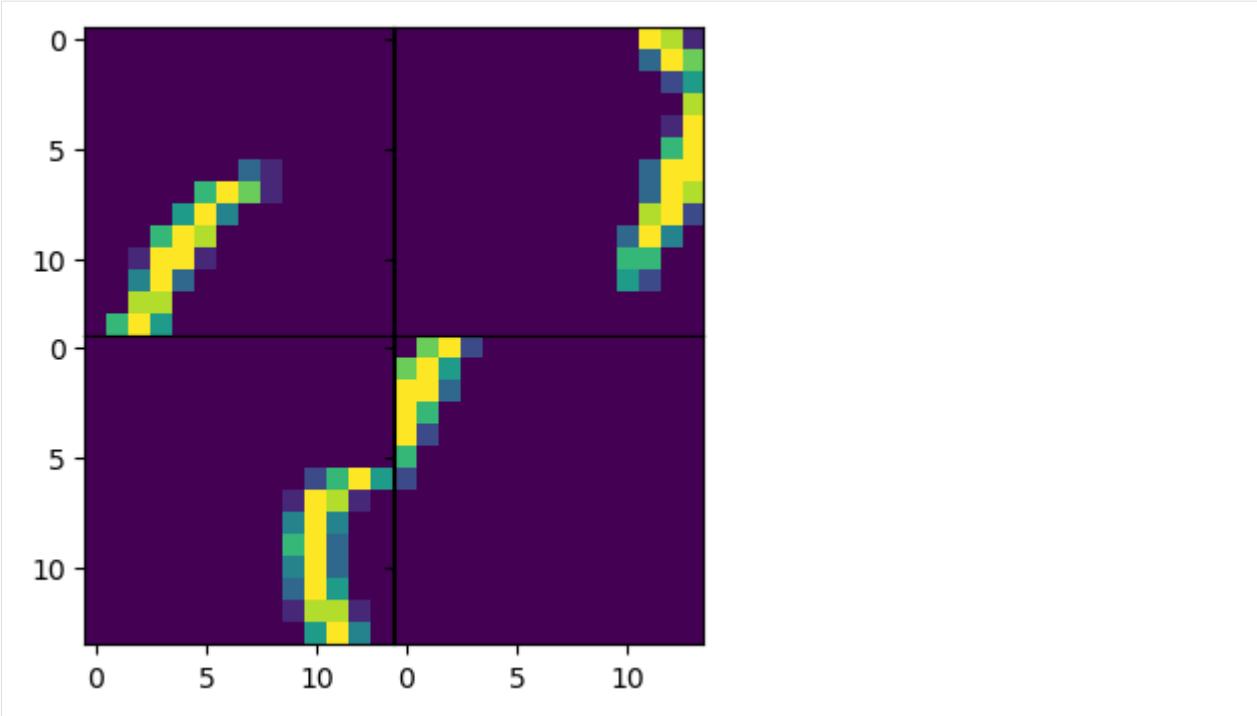
# Chunk the images into 2 pieces
num_pieces = 2

# Select an image from the batch
batch_idx = 0

pieces, random_pieces, perm_list = batch_chunk_image(image_batch, num_pieces=num_pieces)
pieces, random_pieces = pieces.to(device), random_pieces.to(device)

# Plot the original scrambled image.
figs, axs = plt.subplots(num_pieces, num_pieces, figsize=(4,4), sharex=True, sharey=True)
for idx, piece in enumerate(random_pieces[batch_idx]):
    axs[idx // num_pieces, idx % num_pieces].imshow(piece.cpu().squeeze())

plt.subplots_adjust(wspace=0, hspace=0)
plt.show()
```



2.2.2 Training the model

Let's set the hyperparameters for our model.

```
[20]: tau = 0.1 # Temperature parameter.
n_sink_iter = 20 # Number of iterations of Sinkhorn operator.

num_pieces = 2 # Number of pieces each side.
image_size = 28 # Original image size.

in_channels = 1
hidden_channels = 32
kernel_size = 5

epochs = 5
learning_rate = 1e-4
train = False # Set this to false if you only want to evaluate the model
```

```
[21]: model = SinkhornConvNet(
    in_channels=in_channels,
    num_pieces=num_pieces,
    image_size=image_size,
    hidden_channels=hidden_channels,
    kernel_size=kernel_size,
    tau=tau,
    n_sink_iter=n_sink_iter
).to(device)
```

(continues on next page)

(continued from previous page)

```
optimizer = optim.Adam(model.parameters(), learning_rate, eps=1e-8)
```

Next, we train the model. With a small number of pieces, after 5 epochs the network already obtains remarkable accuracy.

```
[22]: if train:
    model.train()
    for epoch in range(epochs):
        sum_loss = 0
        for i, data in enumerate(train_loader):
            inputs, _ = data

            # Chunk the images into pieces
            pieces, random_pieces, _ = batch_chunk_image(inputs, num_pieces)
            pieces, random_pieces = pieces.to(device), random_pieces.to(device)

            # Predict the ordering of the pieces using the Sinkhorn Network
            ordered_pieces, _ = model(random_pieces)

            # Apply MSE Loss
            loss = torch.nn.functional.mse_loss(ordered_pieces, pieces, reduction='sum')

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            sum_loss += loss.item()

        print(f"epoch {epoch} | mean loss {round(sum_loss/len(train_loader.dataset), 4)}")
    )

    if not os.path.exists('./saved_models/'):
        os.mkdir('./saved_models')
    torch.save(model.state_dict(), "./saved_models/model.pth")
```

2.2.2 Evaluating the networks predictions.

To evaluate the network's predicted permutation matrices, we follow the authors in using the Kendall-Tau correlation coefficient. We measure how correlated the predicted list of permutation indices is to the original list of permutation indices.

```
[23]: # Scipy
from scipy.stats import kendalltau

model = SinkhornConvNet(
    in_channels=in_channels,
    num_pieces=num_pieces,
    image_size=image_size,
    hidden_channels=hidden_channels,
    kernel_size=kernel_size,
    tau=tau,
```

(continues on next page)

(continued from previous page)

```

    n_sink_iter=n_sink_iter
).to(device)

model.load_state_dict(torch.load(os.path.join("./saved_models/model.pth")))
model = model.to(device)
model.eval()

with torch.no_grad():

    kendall_taus = []

    for data in test_loader:
        inputs, _ = data
        pieces, random_pieces, perm_list = batch_chunk_image(inputs, num_pieces)
        pieces, random_pieces = pieces.to(device), random_pieces.to(device)

        ordered_pieces, predicted_permutation_matrices = model(random_pieces)

        # Create the list of inverse permutation indices from the predicted
        # permutation matrix.
        predicted_perm_list = predicted_permutation_matrices.transpose(1, 2).max(1)[1]

        # Obtain the Kendall-Tau correlation coefficient for the target
        # and predicted list of permutation matrices.
        for p1, p2 in zip(perm_list, predicted_perm_list):
            kendall_taus.append(
                kendalltau(p1.cpu(), p2.cpu())[0]
            )

    print(f"Mean Kendall-Tau: {np.mean(kendall_taus)}")

/home/apervez/anaconda3/envs/pytorch1.9_2/lib/python3.7/site-packages/torch/nn/
functional.py:718: UserWarning: Named tensors and all their associated APIs are an
experimental feature and subject to change. Please do not use them for anything
important until they are released as stable. (Triggered internally at /opt/conda/
conda-bld/pytorch_1623448224956/work/c10/core/TensorImpl.h:1156.)
return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)

```

Mean Kendall-Tau: 0.9930666666666667

For small numbers of pieces, the correlation coefficient is close to 1! This means our Sinkhorn network is able to correctly find the correct permutation in almost all cases.

Finally, let's inspect model predictions!

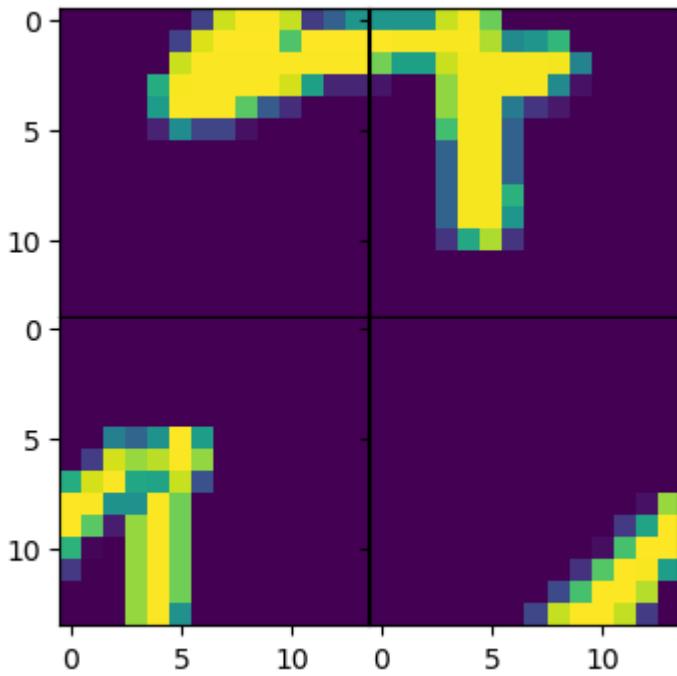
```
[24]: image_batch, _ = next(iter(test_loader))
pieces, random_pieces, perm_list = batch_chunk_image(image_batch, num_pieces)
pieces, random_pieces = pieces.to(device), random_pieces.to(device)

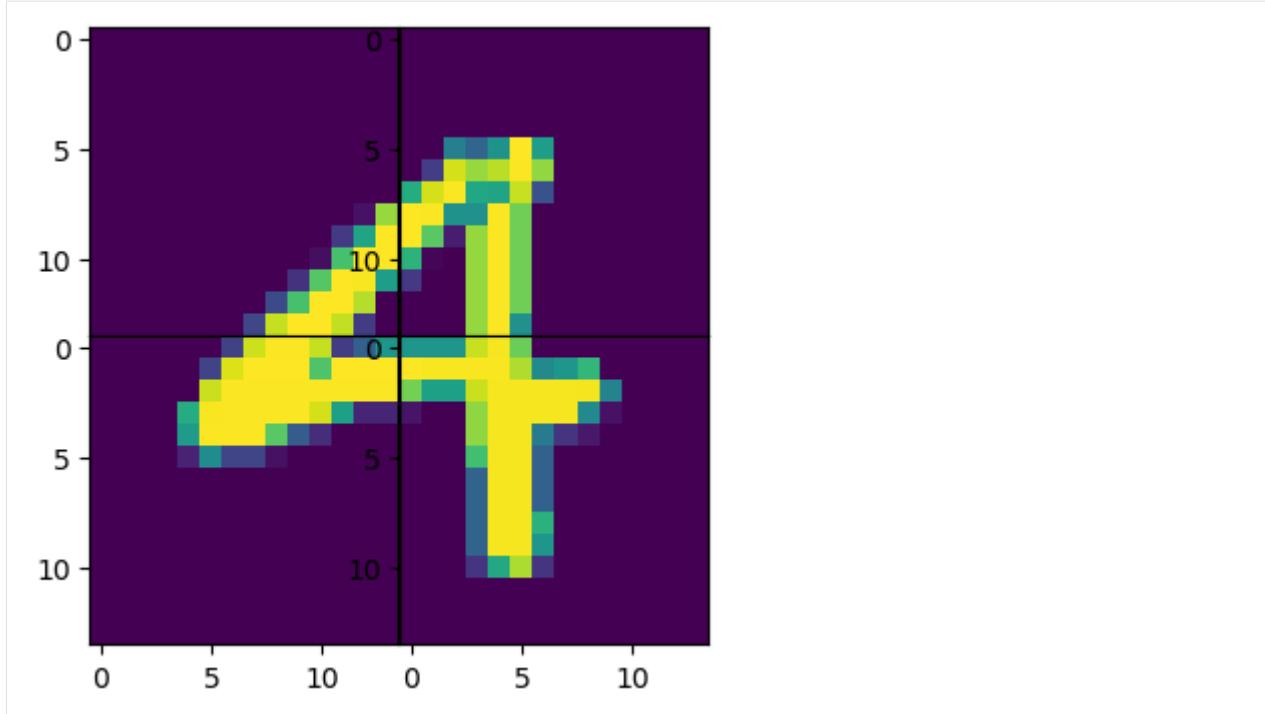
# Make sure we are evaluating!
model.eval()

# Predict the correctly ordered pieces.
predicted_pieces, _ = model(random_pieces)
```

```
[25]: # Select an image from the batch.  
batch_idx = 3
```

```
[26]: # Plot the original scrambled image.  
figs, axs = plt.subplots(num_pieces, num_pieces, figsize=(4,4), sharex=True, sharey=True)  
for idx, piece in enumerate(random_pieces[batch_idx]):  
    axs[idx // num_pieces, idx % num_pieces].imshow(piece.cpu().squeeze())  
plt.subplots_adjust(wspace=0, hspace=0)  
plt.show()  
  
# Plot the predicted reconstructed image.  
figs, axs = plt.subplots(num_pieces, num_pieces, figsize=(4,4))  
for idx, piece in enumerate(predicted_pieces[batch_idx]):  
    axs[idx // num_pieces, idx % num_pieces].imshow(piece.cpu().squeeze())  
plt.subplots_adjust(wspace=0, hspace=0)  
plt.show()
```





4.45 SGA - Graph Sampling for Neural Relational Inference

Notebook:

Author: Adeel Pervez

In this tutorials we show an example of a variational autoencoder model with graph structured latent spaces. This model is called Neural Relational Inference and is described in this [paper](#). This tutorial uses code from the associated code base available [here](#).

The problem dealt by this model is one of predicting particle trajectories. Suppose that we have N interacting particles (say, charges) with some interaction structure (say, attractive/repulsive forces) that are moving about in space. For each particle we are observe its trajectory (say, position and velocity) as it moves about over some period of time T . Each new state in the tracjectory of a particle will depend on the current state (position and velocity) and on the interaction with other particles. Our data consists of a set of N particle trajectories but the actual interactions are unknown. The task of our model is to learn the dynamics of the particles to predict future trajectories given example trajectories only.

Notice that the task of predicting particle dynamics would become easier if we knew the form of the interactions between particles. which we think of as a graph of interactions. Each particle in the system would occupy a node in the graph and the strength of the interaction could be represented by the weight of the graph. The interaction graph could be fed to a neural network alongwith the currently known trajectory which could then predict the next steps of the particles.

However, since in this problem we are not given the interaction graph, the approach taken by Neural Relational Inference is to use the encoder of a variational autoencoder to sample a graph using the given trajectory as input. For this the method uses a graph neural network as encoder.

4.45.1 Graph Network Encoder

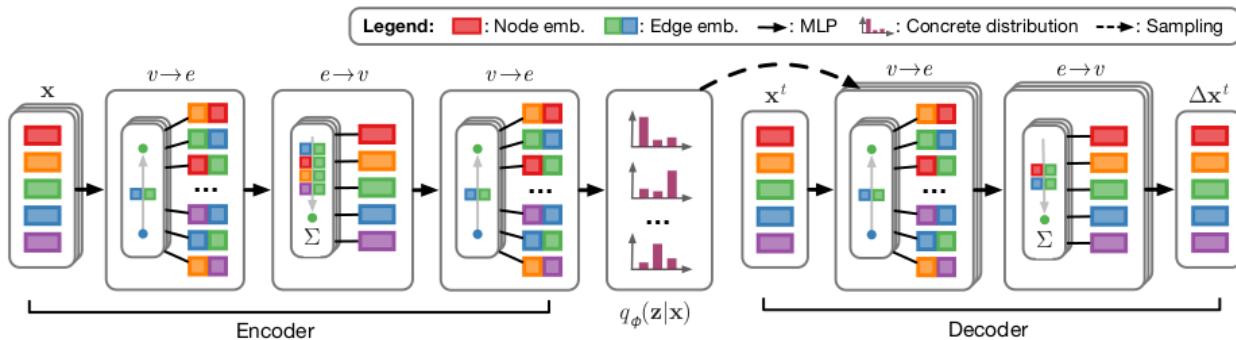
Since graph neural networks already require a graph over which to pass messages, the encoder starts with a fully connected graph with each node representing a particle. The entire trajectory information for a particle is used as the input feature for each node. Node features are transformed and passed over edges and concatenated to form edge features. This process can be repeated to get a deeper network with edge features as the output.

The edge features are then transformed into edge weights (as (unnormalized) log probabilities). Finally from these edge weights we can sample an interaction graph by sample edges according to their weights. This interaction graph then serves as a latent variable z to be used in the decoder where z_{ij} indicates whether edge (i, j) is present in the graph. Since we need to backpropagate into the encoder we relax the edge sampling operation using the Gumbel-Softmax relaxation.

4.45.2 Graph Network Decoder

In the decoder we use another graph network which uses the graph sampled by the encoder and the input trajectory to predict the next step in the trajectory for some predefined number of steps. The models uses a gaussian likelihood loss for the trajectory which is fed into the VAE loss for optimization.

The model structure can be seen in the following figure from the paper linked above.



4.45.3 Neural Relational Inference

Now we build a model to learn the dynamics of a 5 particle system connected by springs. The trajectory data for this system has been generated synthetically using the dynamical equations of motion and a ground truth interaction graph. The code to load the data is given in the `load_data` function in `utils.py`.

As the first step we begin with downloading loading the data.

```
[2]: !wget -O 'nri_springs.zip' https://surfdrive.surf.nl/files/index.php/s/LXV9iJjxfu5jhdD/
--2022-06-21 17:04:34-- https://surfdrive.surf.nl/files/index.php/s/LXV9iJjxfu5jhdD/
Resolving surfdrive.surf.nl (surfdrive.surf.nl)... 145.100.27.67, 2001:610:108:203b:0:
Connecting to surfdrive.surf.nl (surfdrive.surf.nl)|145.100.27.67|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 586393709 (559M) [application/zip]
Saving to: 'nri_springs.zip'

100%[=====] 586,393,709 21.9MB/s   in 21s
```

(continues on next page)

(continued from previous page)

```
2022-06-21 17:04:55 (26.1 MB/s) - 'nri_springs.zip' saved [586393709/586393709]
```

```
[2]: !unzip -u nri_springs.zip -d data
```

```
Archive: nri_springs.zip
```

```
[3]: import time
import argparse
import pickle
import os
import datetime

import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
#import math

from torch.utils.data.dataset import TensorDataset
from torch.utils.data import DataLoader
import torch.nn.functional as F
from torch.autograd import Variable

import torch.optim as optim
from torch.optim import lr_scheduler
import networkx
import matplotlib
import matplotlib.pyplot as plt

try:
    import networkx
except ModuleNotFoundError:
    !pip install --quiet networkx
    import networkx

%matplotlib inline
```

Loading and Examining Data

Next we load the data, convert the arrays into torch tensors and return the loader objects.

```
[4]: def load_data(batch_size=1, suffix=''):
    loc_train = np.load('data/loc_train' + suffix + '.npy')
    vel_train = np.load('data/vel_train' + suffix + '.npy')
    edges_train = np.load('data/edges_train' + suffix + '.npy')

    loc_valid = np.load('data/loc_valid' + suffix + '.npy')
    vel_valid = np.load('data/vel_valid' + suffix + '.npy')
```

(continues on next page)

(continued from previous page)

```

edges_valid = np.load('data/edges_valid' + suffix + '.npy')

loc_test = np.load('data/loc_test' + suffix + '.npy')
vel_test = np.load('data/vel_test' + suffix + '.npy')
edges_test = np.load('data/edges_test' + suffix + '.npy')

# [num_samples, num_timesteps, num_dims, num_atoms]
num_atoms = loc_train.shape[3]

loc_max = loc_train.max()
loc_min = loc_train.min()
vel_max = vel_train.max()
vel_min = vel_train.min()

# Normalize to [-1, 1]
loc_train = (loc_train - loc_min) * 2 / (loc_max - loc_min) - 1
vel_train = (vel_train - vel_min) * 2 / (vel_max - vel_min) - 1

loc_valid = (loc_valid - loc_min) * 2 / (loc_max - loc_min) - 1
vel_valid = (vel_valid - vel_min) * 2 / (vel_max - vel_min) - 1

loc_test = (loc_test - loc_min) * 2 / (loc_max - loc_min) - 1
vel_test = (vel_test - vel_min) * 2 / (vel_max - vel_min) - 1

# Reshape to: [num_sims, num_atoms, num_timesteps, num_dims]
loc_train = np.transpose(loc_train, [0, 3, 1, 2])
vel_train = np.transpose(vel_train, [0, 3, 1, 2])
feat_train = np.concatenate([loc_train, vel_train], axis=3)
edges_train = np.reshape(edges_train, [-1, num_atoms ** 2])
edges_train = np.array((edges_train + 1) / 2, dtype=np.int64)

loc_valid = np.transpose(loc_valid, [0, 3, 1, 2])
vel_valid = np.transpose(vel_valid, [0, 3, 1, 2])
feat_valid = np.concatenate([loc_valid, vel_valid], axis=3)
edges_valid = np.reshape(edges_valid, [-1, num_atoms ** 2])
edges_valid = np.array((edges_valid + 1) / 2, dtype=np.int64)

loc_test = np.transpose(loc_test, [0, 3, 1, 2])
vel_test = np.transpose(vel_test, [0, 3, 1, 2])
feat_test = np.concatenate([loc_test, vel_test], axis=3)
edges_test = np.reshape(edges_test, [-1, num_atoms ** 2])
edges_test = np.array((edges_test + 1) / 2, dtype=np.int64)

feat_train = torch.FloatTensor(feat_train)
edges_train = torch.LongTensor(edges_train)
feat_valid = torch.FloatTensor(feat_valid)
edges_valid = torch.LongTensor(edges_valid)
feat_test = torch.FloatTensor(feat_test)
edges_test = torch.LongTensor(edges_test)

# Exclude self edges
off_diag_idx = np.ravel_multi_index(

```

(continues on next page)

(continued from previous page)

```

    np.where(np.ones((num_atoms, num_atoms)) - np.eye(num_atoms)),
    [num_atoms, num_atoms])
edges_train = edges_train[:, off_diag_idx]
edges_valid = edges_valid[:, off_diag_idx]
edges_test = edges_test[:, off_diag_idx]

train_data = TensorDataset(feat_train, edges_train)
valid_data = TensorDataset(feat_valid, edges_valid)
test_data = TensorDataset(feat_test, edges_test)

train_data_loader = DataLoader(train_data, batch_size=batch_size)
valid_data_loader = DataLoader(valid_data, batch_size=batch_size)
test_data_loader = DataLoader(test_data, batch_size=batch_size)

return train_data_loader, valid_data_loader, test_data_loader, loc_max, loc_min, vel_
max, vel_min

```

We specify the batch size and the data file suffix to load.

[5]: `train_loader, valid_loader, test_loader, _, _, _, _ = load_data(128, "_springs5")`

Let's now examine this data. We get an iterator from the data loader and retrieve the first minibatch. The dataset is in the form of tuples of trajectory information and the ground truth interaction graph.

[6]:

```

(x_sample, rel_sample) = next(iter(train_loader))
print(x_sample.shape)
print(rel_sample.shape)

torch.Size([128, 5, 49, 4])
torch.Size([128, 20])

```

Let's look at the interaction graph first. This dataset consists of trajectories of systems of 5 particles. The interaction graph then specifies for each particle whether or not it interacts with every other particle. For 5 particles this gives us 20 interaction pairs.

[7]:

```

idx=0
print(rel_sample[idx])

tensor([0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0])

```

This interaction is in the form of a list of interactions pairs. We can convert one such list to an interaction graph adjacency matrix as follows. Here we specify that a particle does not interact with itself by setting the diagonal to 0.

[8]:

```

def list_to_adj(rel):
    b = torch.zeros((5*5))
    for i in range(4):
        b[i*5+i+1:(i+1)*5+(i+1)] = rel[i*5:(i+1)*5]
    return b.reshape((5,5))
b=list_to_adj(rel_sample[idx])
print(b.reshape((5,5)))

tensor([[0., 0., 0., 0., 1.],
       [0., 0., 1., 0., 0.],
       [0., 1., 0., 1., 0.],

```

(continues on next page)

(continued from previous page)

```
[0., 0., 1., 0., 0.],
[1., 0., 0., 0., 0.]])
```

We can draw the graph

```
[9]: def show_graph(b):
    g = b.reshape((5,5)).cpu().numpy()
    graph = networkx.from_numpy_array(g)
    networkx.draw(graph, with_labels=True)
    plt.show()
show_graph(b)
```



Let's now examine the trajectory data for the first data point.

```
[10]: x_sample[idx].shape
[10]: torch.Size([5, 49, 4])
```

The shape of the data above specifies that each entry consists of 5 particles with trajectories given for 49 time steps. Furthermore, each state in the trajectory is specified by a 4 dimensional vector. In this case the state is a 2d position and 2d velocity pair specifying the position and velocity of each particle at each time step. We examine the position and velocity of the first particle in the first trajectory for a couple of time steps.

```
[11]: x_sample[idx,0,0:2,:]
[11]: tensor([[-0.1272, -0.0987, -0.3305,  0.1197],
           [-0.1380, -0.0945, -0.3275,  0.1196]])
```

Next we define some hyperparameters.

```
[12]: dims=4
num_atoms=5
timesteps=49
lr=0.0005
```

(continues on next page)

(continued from previous page)

```
temp=0.5
output_var=5e-5

_EPS = 1e-10
```

Recall that we mentioned above that the encoder of the model works on the fully connected graph in order to predict the interaction graph. To pass messages over the fully connected graph, it is useful to define some relational masks specifying which vertices receive messages from which other ones.

```
[13]: def encode_onehot(labels):
    classes = set(labels)
    classes_dict = {c: np.identity(len(classes))[i, :] for i, c in
                    enumerate(classes)}
    labels_onehot = np.array(list(map(classes_dict.get, labels)),
                           dtype=np.int32)
    return labels_onehot

off_diag = np.ones([num_atoms, num_atoms]) - np.eye(num_atoms)

rel_rec = np.array(encode_onehot(np.where(off_diag)[0]), dtype=np.float32)
rel_send = np.array(encode_onehot(np.where(off_diag)[1]), dtype=np.float32)
rel_rec = torch.FloatTensor(rel_rec)
rel_send = torch.FloatTensor(rel_send)
```

We can convert edge features into node features and node features into edge features by passing messages over the fully connected graph using these masks.

```
[14]: print(rel_rec.t(), rel_rec.shape)

tensor([[1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1.,
        1., 0.]] torch.Size([20, 5])
```

For example to convert edge features for the 20 interactions into node features we can multiply the above matrix with the edge feature vector as

```
torch.matmul(rel_rec.t(), x)
```

which collects the messages from all neighboring nodes for each vertex and adds the messages.

Next we define a simple MLP class to be used for the nonlinear feature transformations.

```
[15]: class MLP(nn.Module):
    """Two-layer fully-connected ELU net with batch norm."""

    def __init__(self, n_in, n_hid, n_out, do_prob=0.):
        super(MLP, self).__init__()
```

(continues on next page)

(continued from previous page)

```

    self.fc1 = nn.Linear(n_in, n_hid)
    self.fc2 = nn.Linear(n_hid, n_out)
    self.bn = nn.BatchNorm1d(n_out)
    self.dropout_prob = do_prob

    self.init_weights()

def init_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Linear):
            nn.init.xavier_normal_(m.weight.data)
            m.bias.data.fill_(0.1)
        elif isinstance(m, nn.BatchNorm1d):
            m.weight.data.fill_(1)
            m.bias.data.zero_()

def batch_norm(self, inputs):
    x = inputs.view(inputs.size(0) * inputs.size(1), -1)
    x = self.bn(x)
    return x.view(inputs.size(0), inputs.size(1), -1)

def forward(self, inputs):
    # Input shape: [num_sims, num_things, num_features]
    x = F.elu(self.fc1(inputs))
    x = F.dropout(x, self.dropout_prob, training=self.training)
    x = F.elu(self.fc2(x))
    return self.batch_norm(x)

```

Encoder

Next we specify the VAE encoder as a graph neural network. For each particle we construct a single feature vector by using the entire trajectory information. We include three graph neural network layers that convert the node features into edge features into node features and finally into edge features. The final edge features are then used to parameterize the edge probabilities for the graph sampling operation.

[16]:

```

class MLPEncoder(nn.Module):
    def __init__(self, n_in, n_hid, n_out=2, do_prob=0.,):
        super(MLPEncoder, self).__init__()

        self.mlp1 = MLP(n_in, n_hid, n_hid, do_prob)
        self.mlp2 = MLP(n_hid * 2, n_hid, n_hid, do_prob)
        self.mlp3 = MLP(n_hid, n_hid, n_hid, do_prob)
        self.mlp4 = MLP(n_hid * 3, n_hid, n_hid, do_prob)

        self.fc_out = nn.Linear(n_hid, n_out)
        self.init_weights()

    def init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):

```

(continues on next page)

(continued from previous page)

```

nn.init.xavier_normal(m.weight.data)
m.bias.data.fill_(0.1)

def edge2node(self, x, rel_rec, rel_send):
    # NOTE: Assumes that we have the same graph across all samples.
    incoming = torch.matmul(rel_rec.t(), x)
    return incoming / incoming.size(1)

def node2edge(self, x, rel_rec, rel_send):
    # NOTE: Assumes that we have the same graph across all samples.
    receivers = torch.matmul(rel_rec, x)
    senders = torch.matmul(rel_send, x)
    edges = torch.cat([senders, receivers], dim=2)
    return edges

def forward(self, inputs, rel_rec, rel_send):
    # Input shape: [num_sims, num_atoms, num_timesteps, num_dims]
    x = inputs.view(inputs.size(0), inputs.size(1), -1)
    # New shape: [num_sims, num_atoms, num_timesteps*num_dims]

    x = self.mlp1(x)  # 2-layer ELU net per node

    x = self.node2edge(x, rel_rec, rel_send)
    x = self.mlp2(x)
    x_skip = x

    x = self.edge2node(x, rel_rec, rel_send)
    x = self.mlp3(x)
    x = self.node2edge(x, rel_rec, rel_send)
    x = torch.cat((x, x_skip), dim=2)  # Skip connection
    x = self.mlp4(x)

    return self.fc_out(x)

```

Decoder

In the decoder we take the initial state and attempt to predict the remaining trajectory. To predict a single time step the function `single_step_forward` takes as input the current state and the proposed interaction graph. From the states of the nodes as node features we produce edge features by passing messages over all neighbors and then zero out the messages corresponding to edges not present in the proposed graph. The edge messages are then sent to the corresponding nodes after which we apply an MLP to compute the next prediction as a difference with the current state. The process is then repeated for all timesteps.

```
[17]: class MLPDecoder(nn.Module):
    """MLP decoder module."""

    def __init__(self, n_in_node, edge_types, msg_hid, msg_out, n_hid,
                 do_prob=0.):
        super(MLPDecoder, self).__init__()

        self.msg_fc1 = (nn.Linear(2 * n_in_node, msg_hid))
```

(continues on next page)

(continued from previous page)

```

self.msg_fc2 = (nn.Linear(msg_hid, msg_out))
self.msg_out_shape = msg_out

self.out_fc1 = nn.Linear(n_in_node + msg_out, n_hid)
self.out_fc2 = nn.Linear(n_hid, n_hid)
self.out_fc3 = nn.Linear(n_hid, n_in_node)

print('Using learned interaction net decoder.')

self.dropout_prob = do_prob

def single_step_forward(self, single_timestep_inputs, rel_rec, rel_send,
                      single_timestep_rel_type):

    # single_timestep_inputs has shape
    # [batch_size, num_timesteps, num_atoms, num_dims]

    # single_timestep_rel_type has shape:
    # [batch_size, num_timesteps, num_atoms*(num_atoms-1), num_edge_types]

    # Node2edge
    receivers = torch.matmul(rel_rec, single_timestep_inputs)
    senders = torch.matmul(rel_send, single_timestep_inputs)
    pre_msg = torch.cat([senders, receivers], dim=-1)

    msg = F.relu(self.msg_fc1(pre_msg))
    msg = F.dropout(msg, p=self.dropout_prob)
    msg = F.relu(self.msg_fc2(msg))
    msg = msg * single_timestep_rel_type[:, :, :, 1:2]

    # Aggregate all msgs to receiver
    agg_msgs = msg.transpose(-2, -1).matmul(rel_rec).transpose(-2, -1)
    agg_msgs = agg_msgs.contiguous()

    # Skip connection
    aug_inputs = torch.cat([single_timestep_inputs, agg_msgs], dim=-1)

    # Output MLP
    pred = F.dropout(F.relu(self.out_fc1(aug_inputs)), p=self.dropout_prob)
    pred = F.dropout(F.relu(self.out_fc2(pred)), p=self.dropout_prob)
    pred = self.out_fc3(pred)

    # Predict position/velocity difference
    return single_timestep_inputs + pred

def forward(self, inputs, rel_type, rel_rec, rel_send, pred_steps=1):
    # NOTE: Assumes that we have the same graph across all samples.

    inputs = inputs.transpose(1, 2).contiguous()

    sizes = [rel_type.size(0), inputs.size(1), rel_type.size(1),
             rel_type.size(2)]

```

(continues on next page)

(continued from previous page)

```

rel_type = rel_type.unsqueeze(1).expand(sizes)

time_steps = inputs.size(1)
assert (pred_steps <= time_steps)
preds = []

# initial step
last_pred = inputs[:, 0:1, :, :]
# NOTE: Assumes rel_type is constant (i.e. same across all time steps).
curr_rel_type = rel_type[:, 0:1, :, :]

# Run n prediction steps
for step in range(0, pred_steps):
    last_pred = self.single_step_forward(last_pred, rel_rec, rel_send,
                                         curr_rel_type)
    preds.append(last_pred)

sizes = [preds[0].size(0), preds[0].size(1) * pred_steps,
        preds[0].size(2), preds[0].size(3)]

#output = Variable(torch.zeros(sizes))
output = torch.zeros(sizes)
if inputs.is_cuda:
    output = output.cuda()

# Re-assemble correct timeline
for i in range(len(preds)):
    output[:, i:i+1, :, :] = preds[i]

pred_all = output[:, :(inputs.size(1) - 1), :, :]

return pred_all.transpose(1, 2).contiguous()

```

Now we make a few helper arrays to specify the sending and receiving nodes in a message passing step. These simply correspond to the neighbors in a fully connected graph.

[18]:

```

# Generate off-diagonal interaction graph
off_diag = np.ones([num_atoms, num_atoms]) - np.eye(num_atoms)

rel_rec = np.array(encode_onehot(np.where(off_diag)[0]), dtype=np.float32)
rel_send = np.array(encode_onehot(np.where(off_diag)[1]), dtype=np.float32)
rel_rec = torch.FloatTensor(rel_rec)
rel_send = torch.FloatTensor(rel_send)

```

[19]: #edge weights have dim 2

```

encoder = MLPEncoder(timesteps * dims, 256, 2)
decoder = MLPDecoder(n_in_node=dims,
                     edge_types=2,
                     msg_hid=256,
                     msg_out=256,

```

(continues on next page)

(continued from previous page)

```
n_hid=256, )
```

Using learned interaction net decoder.

```
/home/apervez/anaconda3/envs/pytorch1.9_2/lib/python3.7/site-packages/ipykernel_launcher.py:16: UserWarning: nn.init.xavier_normal is now deprecated in favor of nn.init.xavier_normal_.  
app.launch_new_instance()
```

[20]:

```
encoder.cuda()  
decoder.cuda()  
rel_rec = rel_rec.cuda()  
rel_send = rel_send.cuda()
```

[21]:

```
optimizer = optim.Adam(list(encoder.parameters()) + list(decoder.parameters()),  
                      lr=lr)  
train_loader, valid_loader, test_loader, loc_max, loc_min, vel_max, vel_min = load_data(  
    128, "_springs5")
```

Discrete Sampling

We will sample graphs by selecting a subset of the fully connected graph using the weights output by the encoder by using Gumbel softmax. We define the Gumbel softmax routines below as from the earlier tutorial.

[22]:

```
def my_softmax(input, axis=1):  
    trans_input = input.transpose(axis, 0).contiguous()  
    soft_max_1d = F.softmax(trans_input)  
    return soft_max_1d.transpose(axis, 0)  
  
def sample_gumbel(shape, eps=1e-10):  
    U = torch.rand(shape).float()  
    return - torch.log(eps - torch.log(U + eps))  
  
def gumbel_softmax_sample(logits, tau=1, eps=1e-10):  
    gumbel_noise = sample_gumbel(logits.size(), eps=eps)  
    if logits.is_cuda:  
        gumbel_noise = gumbel_noise.cuda()  
    y = logits + Variable(gumbel_noise)  
    return my_softmax(y / tau, axis=-1)  
  
def gumbel_softmax(logits, tau=1, hard=False, eps=1e-10):  
    y_soft = gumbel_softmax_sample(logits, tau=tau, eps=eps)  
    if hard:  
        shape = logits.size()  
        _, k = y_soft.data.max(-1)  
        y_hard = torch.zeros(*shape)  
        if y_soft.is_cuda:  
            y_hard = y_hard.cuda()  
        y_hard = y_hard.zero_().scatter_(-1, k.view(shape[:-1] + (1,)), 1.0)
```

(continues on next page)

(continued from previous page)

```

y = Variable(y_hard - y_soft.data) + y_soft
else:
    y = y_soft
return y

```

Next we define some helper functions for computing accuracies, prediction loss and KL divergence.

[23]:

```

def kl_categorical_uniform(preds, num_atoms, num_edge_types, add_const=False,
                           eps=1e-16):
    kl_div = preds * torch.log(preds + eps)
    if add_const:
        const = np.log(num_edge_types)
        kl_div += const
    return kl_div.sum() / (num_atoms * preds.size(0))

def nll_gaussian(preds, target, variance, add_const=False):
    neg_log_p = ((preds - target) ** 2 / (2 * variance))
    if add_const:
        const = 0.5 * np.log(2 * np.pi * variance)
        neg_log_p += const
    return neg_log_p.sum() / (target.size(0) * target.size(1))

def edge_accuracy(preds, target):
    _, preds = preds.max(-1)
    correct = preds.float().data.eq(
        target.float().data.view_as(preds)).cpu().sum()
    return np.float(correct) / (target.size(0) * target.size(1))

```

We can now train the model. This runs the encoder to get the latent graph parameters and samples the edges using the `gumbel_softmax` function to get a latent graph which is then passed to the decoder. We use the uniform categorical prior to compute the KL divergence for the VAE loss and a Gaussian likelihood loss with a fixed variance for the predictions.

[24]:

```

def train(epoch, best_val_loss):
    t = time.time()
    nll_train = []
    acc_train = []
    kl_train = []
    mse_train = []

    encoder.train()
    decoder.train()
    #scheduler.step()
    for batch_idx, (data, relations) in enumerate(train_loader):

        #if args.cuda:
        data, relations = data.cuda(), relations.cuda()

```

(continues on next page)

(continued from previous page)

```

#data, relations = Variable(data), Variable(relations)

optimizer.zero_grad()

logits = encoder(data, rel_rec, rel_send)
edges = gumbel_softmax(logits, tau=temp, hard=False)
prob = my_softmax(logits, -1)

output = decoder(data, edges, rel_rec, rel_send, timesteps)

target = data[:, :, 1:, :]

loss_nll = nll_gaussian(output, target, output_var)

loss_kl = kl_categorical_uniform(prob, num_atoms, 2)

loss = loss_nll + loss_kl

acc = edge_accuracy(logits, relations)
acc_train.append(acc)

loss.backward()
optimizer.step()

mse_train.append(F.mse_loss(output, target).item())
nll_train.append(loss_nll.item())
kl_train.append(loss_kl.item())

nll_val = []
acc_val = []
kl_val = []
mse_val = []

encoder.eval()
decoder.eval()
for batch_idx, (data, relations) in enumerate(valid_loader):
    #if args.cuda:
    data, relations = data.cuda(), relations.cuda()

    logits = encoder(data, rel_rec, rel_send)
    edges = gumbel_softmax(logits, tau=temp, hard=True)
    prob = my_softmax(logits, -1)

    # validation output uses teacher forcing
    output = decoder(data, edges, rel_rec, rel_send, timesteps)

    target = data[:, :, 1:, :]
    loss_nll = nll_gaussian(output, target, output_var)

    loss_kl = kl_categorical_uniform(prob, num_atoms, 2)

    acc = edge_accuracy(logits, relations)

```

(continues on next page)

(continued from previous page)

```

    acc_val.append(acc)

    mse_val.append(F.mse_loss(output, target).item())
    nll_val.append(loss_nll.item())
    kl_val.append(loss_kl.item())

    print('Epoch: {:04d}'.format(epoch),
          'nll_train: {:.10f}'.format(np.mean(nll_train)),
          'kl_train: {:.10f}'.format(np.mean(kl_train)),
          'mse_train: {:.10f}'.format(np.mean(mse_train)),
          'acc_train: {:.10f}'.format(np.mean(acc_train)),
          'nll_val: {:.10f}'.format(np.mean(nll_val)),
          'kl_val: {:.10f}'.format(np.mean(kl_val)),
          'mse_val: {:.10f}'.format(np.mean(mse_val)),
          'acc_val: {:.10f}'.format(np.mean(acc_val)),
          'time: {:.4f}s'.format(time.time() - t))
    return np.mean(nll_val)

```

[25]:

```

t_total = time.time()
best_val_loss = np.inf
best_epoch = 0
for epoch in range(10):
    val_loss = train(epoch, best_val_loss)
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_epoch = epoch

/home/apervez/anaconda3/envs/pytorch1.9_2/lib/python3.7/site-packages/ipykernel_launcher.
└─py:3: UserWarning: Implicit dimension choice for softmax has been deprecated. Change
└─the call to include dim=X as an argument.
    This is separate from the ipykernel package so we can avoid doing imports until

Epoch: 0000 nll_train: 82207.3378531610 kl_train: -0.4316625203 mse_train: 0.0428163237
└─acc_train: 0.5102473625 nll_val: 13553.7608163568 kl_val: -0.5440562605 mse_val: 0.
└─0070592507 acc_val: 0.5207624604 time: 39.0780s
Epoch: 0001 nll_train: 10482.0866405750 kl_train: -0.8068965521 mse_train: 0.0054594203
└─acc_train: 0.5672370524 nll_val: 9608.4652764043 kl_val: -0.9848571722 mse_val: 0.
└─0050044091 acc_val: 0.6150959256 time: 38.8682s
Epoch: 0002 nll_train: 7622.5237534467 kl_train: -0.9681852530 mse_train: 0.0039700646
└─acc_train: 0.6575051950 nll_val: 7999.2846617880 kl_val: -0.9407296845 mse_val: 0.
└─0041662942 acc_val: 0.6889833861 time: 38.9219s
Epoch: 0003 nll_train: 6435.7686271180 kl_train: -0.8848858903 mse_train: 0.0033519629
└─acc_train: 0.7282261029 nll_val: 6645.1351859177 kl_val: -0.8726646719 mse_val: 0.
└─0034610081 acc_val: 0.7700009889 time: 38.9155s
Epoch: 0004 nll_train: 5131.3655265945 kl_train: -0.7572590768 mse_train: 0.0026725863
└─acc_train: 0.8130400815 nll_val: 4324.9007367484 kl_val: -0.6718307175 mse_val: 0.
└─0022525526 acc_val: 0.8733831092 time: 38.8255s
Epoch: 0005 nll_train: 3077.5661589874 kl_train: -0.5434278285 mse_train: 0.0016028991
└─acc_train: 0.8900433584 nll_val: 3308.9670224733 kl_val: -0.4739881709 mse_val: 0.
└─0017234204 acc_val: 0.9016811709 time: 38.8303s

```

(continues on next page)

(continued from previous page)

```

Epoch: 0006 nll_train: 2414.1095929158 kl_train: -0.4157265660 mse_train: 0.0012573488 ↵
↳ acc_train: 0.9119279492 nll_val: 2641.4135803995 kl_val: -0.3809240079 mse_val: 0. ↳
↳ 0013757363 acc_val: 0.9172320016 time: 38.7057s
Epoch: 0007 nll_train: 2069.5431726168 kl_train: -0.3398666841 mse_train: 0.0010778871 ↵
↳ acc_train: 0.9245336477 nll_val: 2234.7316940887 kl_val: -0.3186675578 mse_val: 0. ↳
↳ 0011639228 acc_val: 0.9260087025 time: 38.7296s
Epoch: 0008 nll_train: 1849.1151079339 kl_train: -0.2847137487 mse_train: 0.0009630808 ↵
↳ acc_train: 0.9329048114 nll_val: 2150.2909862663 kl_val: -0.2708904545 mse_val: 0. ↳
↳ 0011199433 acc_val: 0.9309285997 time: 38.5185s
Epoch: 0009 nll_train: 1702.6682693989 kl_train: -0.2453164980 mse_train: 0.0008868064 ↵
↳ acc_train: 0.9392786925 nll_val: 1745.5321894778 kl_val: -0.2299301707 mse_val: 0. ↳
↳ 0009091314 acc_val: 0.9383850870 time: 38.7678s

```

Visualizing Discovered Graphs

We can now visualize the actual and predicted interaction graphs for some examples.

```
[26]: data,relations = next(iter(valid_loader))
data=data.cuda()
relations=relations.cuda()
```

```
[27]: logits = encoder(data, rel_rec, rel_send)
_, rel = logits.max(-1)
```

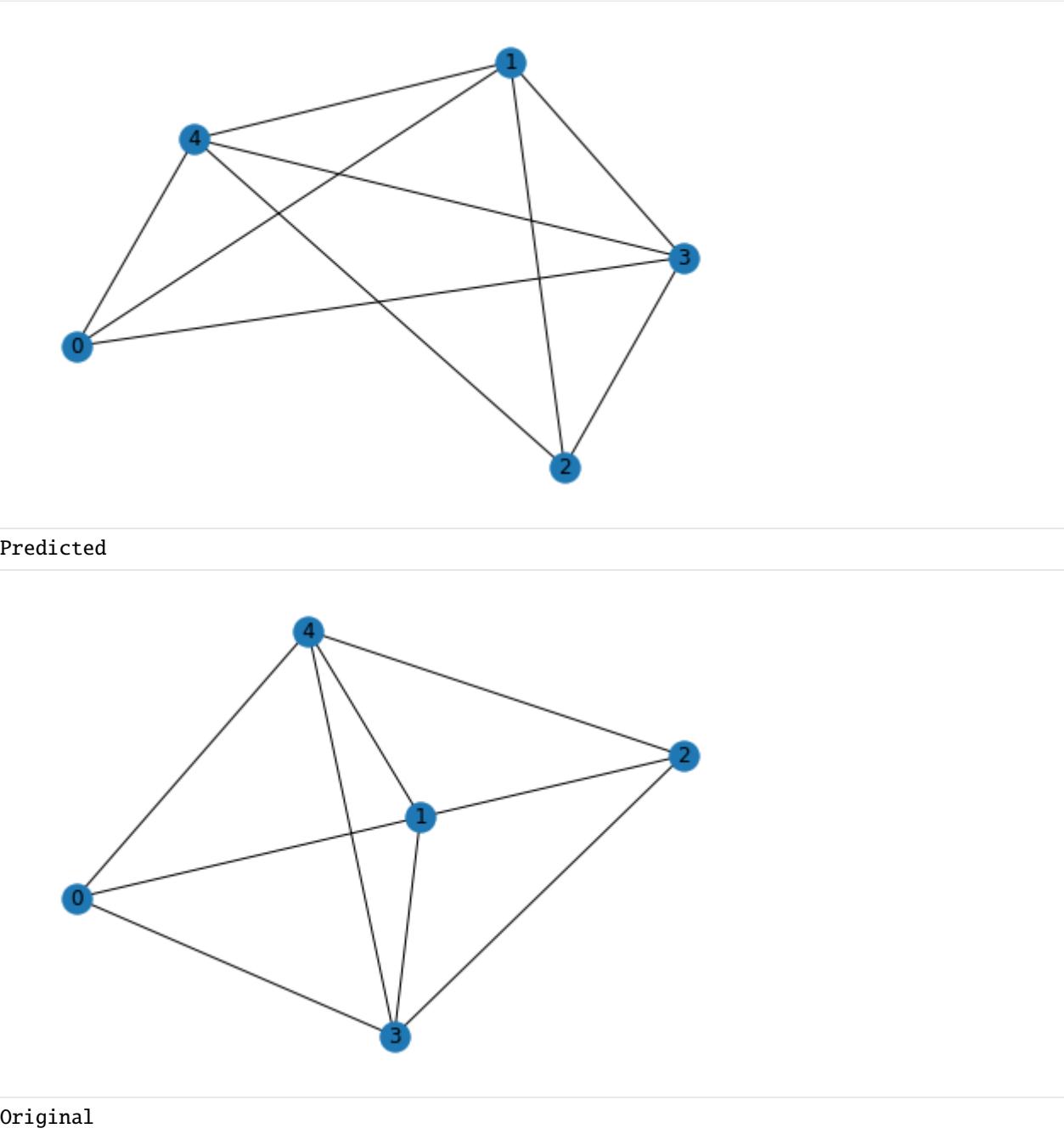
```
[28]: print(rel[0])
print(relations[0])

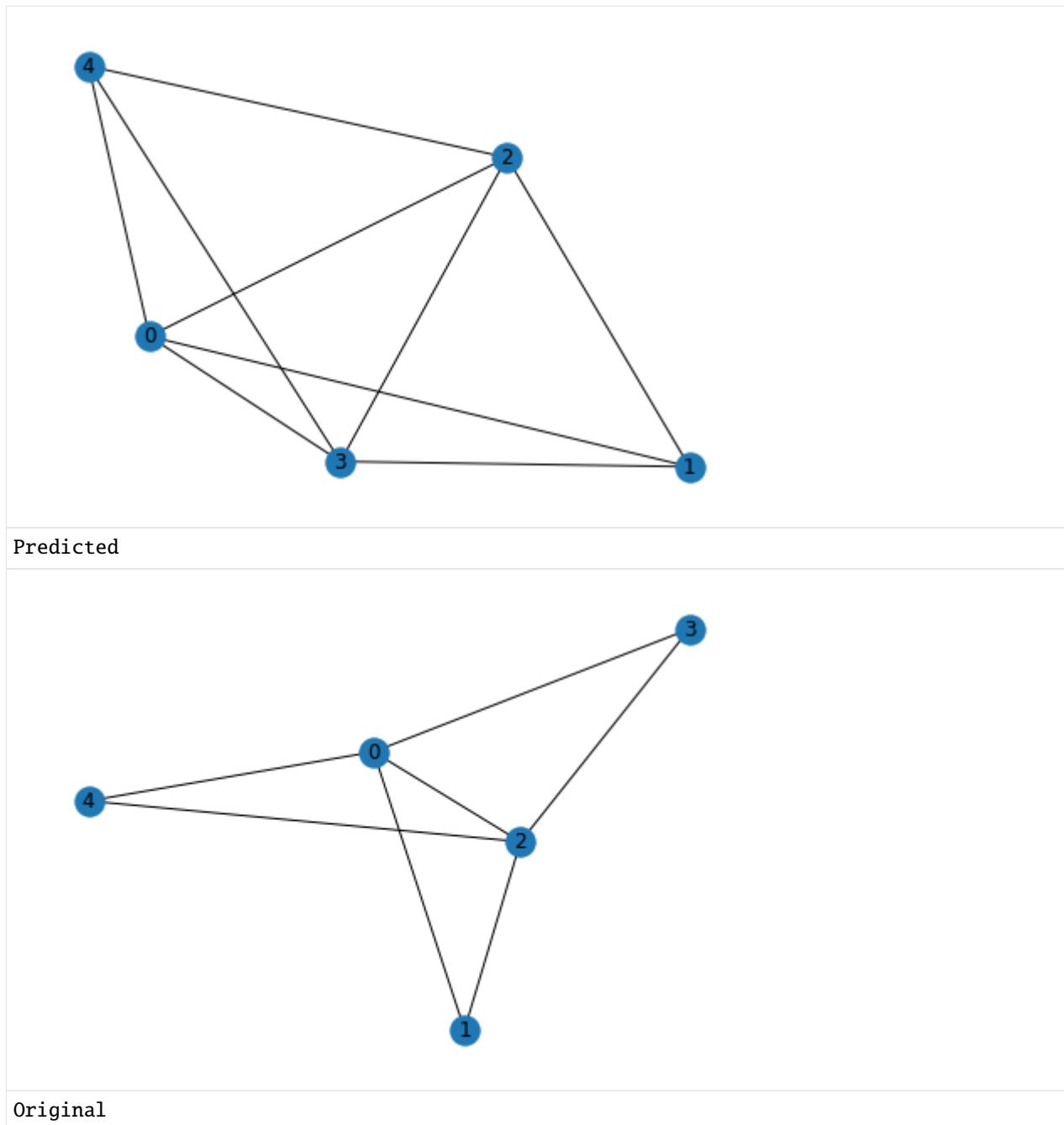
tensor([1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], device='cuda:0')
tensor([1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], device='cuda:0')
```

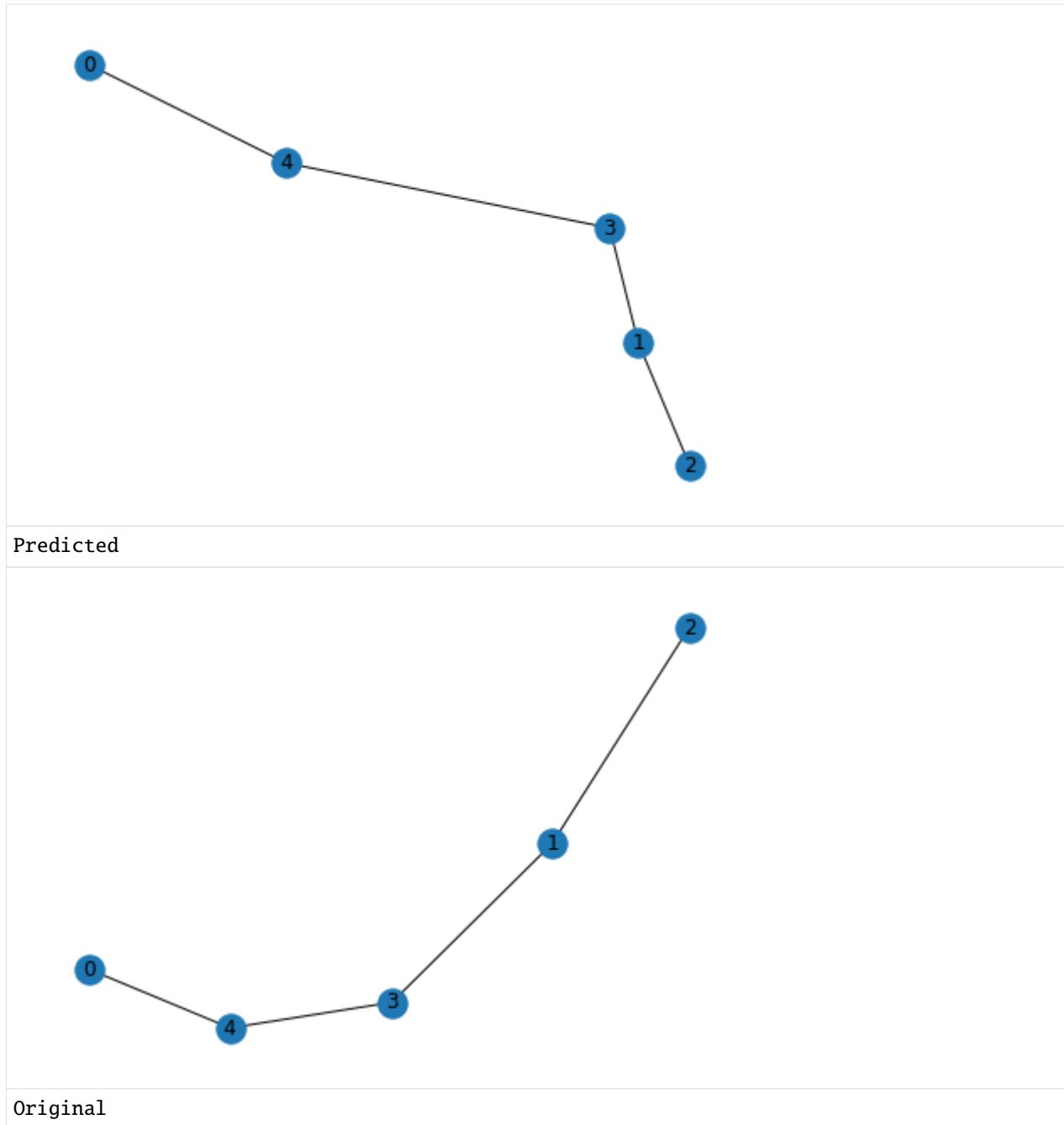
```
[29]: for i in range(5):
    g_act = list_to_adj(relations[i])
    g_pred = list_to_adj(rel[i])

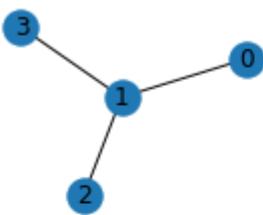
    print("Original")
    show_graph(g_act)
    print("Predicted")
    show_graph(g_pred)
```

Original





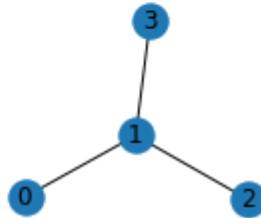




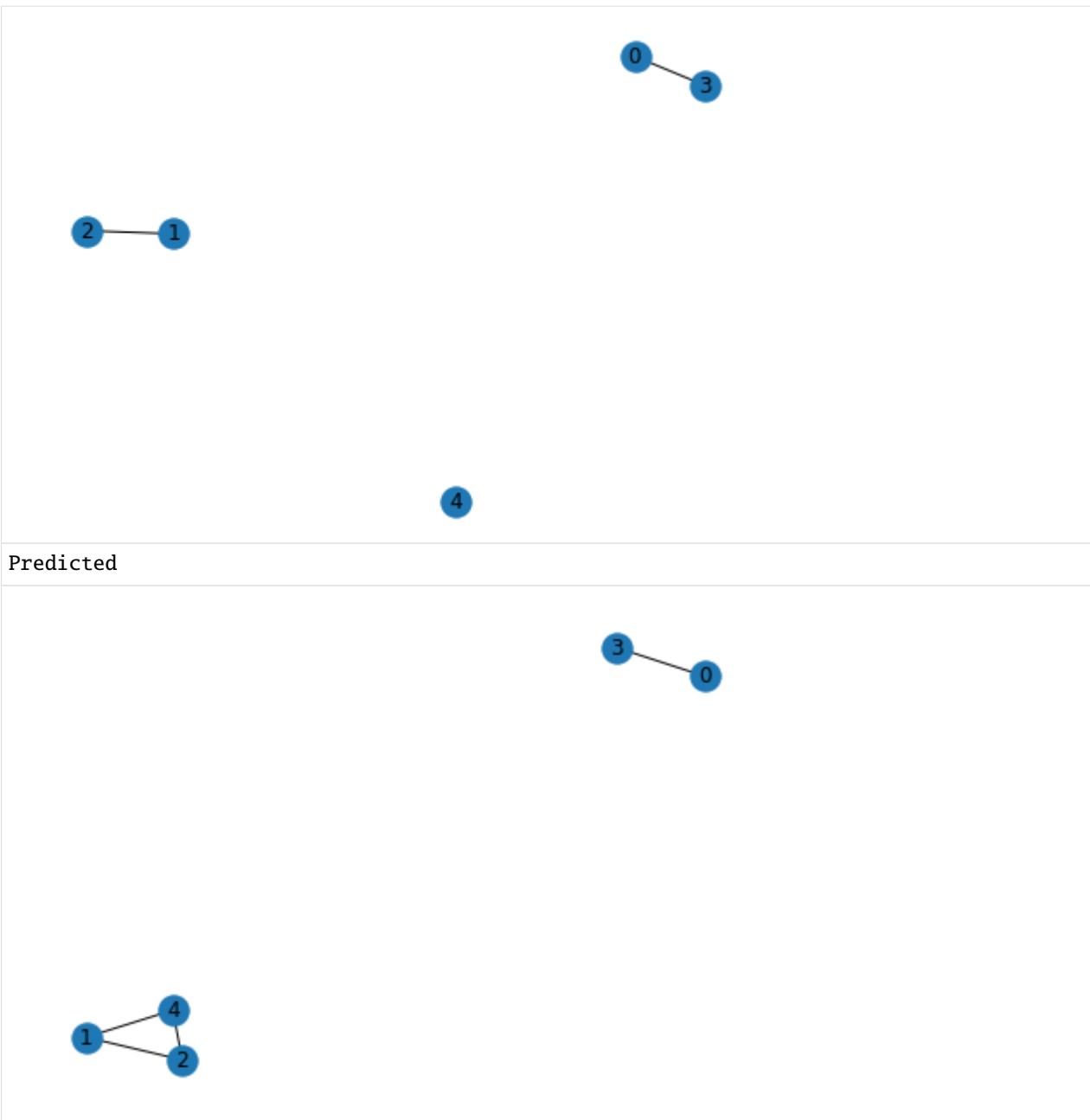
4

Predicted

4



Original



4.45.4 References

Neural Relational Inference for Interacting Systems