

# Deep Learning Lab

**Instructor:** Kazuki Irie

**TAs:** Róbert Csordás, Aditya Ramesh

[kazuki.irie@usi.ch](mailto:kazuki.irie@usi.ch), [robert.csordas@idsia.ch](mailto:robert.csordas@idsia.ch), [aditya.ramesh@idsia.ch](mailto:aditya.ramesh@idsia.ch)

The Swiss AI Lab IDSIA, USI & SUPSI

First version: 19.09.2022, Last modified: 11.12.2022/23:26  
Fall 2022, USI.

- 1. Workflow Overview**
- 2. Introduction to Tools**
  1. Basic Tools
  2. Implementation of Systems
- 3. Fundamental building blocks**
  1. Feed-forward NNs, Convolutional NNs, Residual NNs
  2. Recurrent NNs and LSTM
  3. Attention, Self-attention, Transformers
- 4. Building models**
- 5. Practical tricks and methods for training**
- 6. Final words and Outlook**

## Structure and objectives:

- Acquire hands-on experience - *Lab!*
- Focus on practical aspects of machine learning/deep learning  
(≈ 50% lectures, 50% exercises)
- (Originally designed to) Complement the lecture *Machine Learning*
- But self-contained.

## Introduction to:

- Framework, tools, implementation (PyTorch)
- Basic **practicalities** of deep learning: workflow, hyper-parameter tuning, practical tricks...
- ...using various types of problems and model architectures
- Learning to autonomously search for knowledge/solution to problems

# Course Logistics

## Time and location:

- **Mondays from 10:30 to 12:00** (no break in principle)
- Lecture room: **C1.03** Est Campus

<https://www.desk.usi.ch/en/lugano-campus-map-access-facilities>

## Time and location:

- **Mondays from 10:30 to 12:00** (no break in principle)
- Lecture room: **C1.03** Est Campus

<https://www.desk.usi.ch/en/lugano-campus-map-access-facilities>

## Format:

- 3 ETCS credits for Master (2 for PhD).
- Lectures + guided exercises: 4 of these exercises are **assignments**.
- **Live streaming** on Microsoft Teams “Deep Learning Lab 2022”  
Use your USI account and access code: **ezdr8cr**  
You can raise your hand for questions (but it may take some time for me to notice).
- **All lectures/sessions will be recorded.**

# Course Logistics

## Time and location:

- **Mondays from 10:30 to 12:00** (no break in principle)
- Lecture room: **C1.03** Est Campus

<https://www.desk.usi.ch/en/lugano-campus-map-access-facilities>

## Format:

- 3 ETCS credits for Master (2 for PhD).
- Lectures + guided exercises: 4 of these exercises are **assignments**.
- **Live streaming** on Microsoft Teams “Deep Learning Lab 2022”  
Use your USI account and access code: **ezdr8cr**  
You can raise your hand for questions (but it may take some time for me to notice).
- **All lectures/sessions will be recorded.**

## Course materials:

- All course materials (slides, exercise sheets) will be uploaded on **iCorsi3**.  
Slides will get gradually updated (pls. check the date/version).
- Videos will be uploaded on **Panopto**.

## Course Logistics (cont'd)

### No more special disposition related to COVID at USI

- It is your decision to opt for wearing a mask in the classroom (or not).

See the official announcement: <https://www.usi.ch/en/feeds/13812>

# Grading and Rules

## Grades:

- Entirely determined by your performance on the **4 assignments!**
- Difficulty/length of assignments gradually increases.
- Final grade: weighted average (15%, 20%, 25%, and 40%)
- All assignments must be solved using Python and PyTorch.  
(Solution in other languages will not be accepted).
- More generally: **please respect rules specified in the assignments!**

# Grading and Rules

## Grades:

- Entirely determined by your performance on the **4 assignments!**
- Difficulty/length of assignments gradually increases.
- Final grade: weighted average (15%, 20%, 25%, and 40%)
- All assignments must be solved using Python and PyTorch.  
(Solution in other languages will not be accepted).
- More generally: **please respect rules specified in the assignments!**

## Deadlines:

- Each assignment has its dedicated presentation session:
  - I will shortly present the contents of the assignments.
  - TA(s) and I stay available for your questions in the remaining time.
- The assignments must be submitted **within 2 weeks** from the session:  
**Sunday evening at 10:00 PM** (except for the last assignment).
- All assignment/exercise sheets will be available on iCorsi3 (soon!).

# Grading and Rules (cont'd)

## Late submission policy:

- Deadline Sunday 10:00 PM.
- **10 min grace period:** no reduction if submitted between 10:00 - 10:10 PM.
- **-30% to your score** if submitted within 3 days after the deadline, i.e. between Sunday 10:11 PM - Wednesday 10:00 PM.
- 0 point if submitted later.

# Grading and Rules (cont'd)

## Late submission policy:

- Deadline Sunday 10:00 PM.
- **10 min grace period:** no reduction if submitted between 10:00 - 10:10 PM.
- **-30% to your score** if submitted within 3 days after the deadline, i.e. between Sunday 10:11 PM - Wednesday 10:00 PM.
- 0 point if submitted later.

## Collaboration policy:

- You may discuss with each other, **but**:
- You must individually submit **your own solution**.
- In case of plagiarism: score of 0 for everyone involved (more on this later)

# Grading and Rules (cont'd)

## Late submission policy:

- Deadline Sunday 10:00 PM.
- **10 min grace period:** no reduction if submitted between 10:00 - 10:10 PM.
- **-30% to your score** if submitted within 3 days after the deadline, i.e. between Sunday 10:11 PM - Wednesday 10:00 PM.
- 0 point if submitted later.

## Collaboration policy:

- You may discuss with each other, **but**:
- You must individually submit **your own solution**.
- In case of plagiarism: score of 0 for everyone involved (more on this later)

## Course attendance:

- Not mandatory.
- But obviously: highly recommended to attend/watch all lectures!
- Ask questions about the assignments in the dedicated Q/A sessions, not the day before the deadline!

- Plagiarism can have very serious and unpleasant consequences.
  - See USI's study regulation Art. 38. 1 and 2.  
<https://content.usi.ch/sites/default/files/storage/attachments/inf-inf-study-regulations-faculty-informatics-2013-2014-bachelor-master.pdf>
- Typical cases: lines of code or textual answers copied from the internet or from a colleague with minor modifications.
- **It is strictly forbidden to exchange your code with your colleagues in any form.**
  - The whole point of this course is about learning to write your own code.
  - You are allowed to use code presented in the lecture as your starting point. But nothing else should be copied.
  - Unfortunately: last year 14 students out of 60 got 0 on Assignment 3 (especially unfortunate for good students who actually solved the problem but got 0 because they shared their code to others who plagiarized it).

# Grading and Rules (cont'd)

## Reports:

- For assignments, you will have to submit reports.
- Please use **LaTeX** to write your report.
- If you do not know **LATEX** yet, it's an opportunity to learn it!

*Learn LaTeX in 30 min:*

[https://www.overleaf.com/learn/latex/Learn\\_LaTeX\\_in\\_30\\_minutes](https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes)

- Should be useful for your future reports/articles/papers/theses (not only for this course!)
- Useful online tool, **Overleaf**: <https://www.overleaf.com/>

# Grading and Rules (cont'd)

## Reports:

- For assignments, you will have to submit reports.
- Please use **LaTeX** to write your report.
- If you do not know **LaTeX** yet, it's an opportunity to learn it!

*Learn LaTeX in 30 min:*

[https://www.overleaf.com/learn/latex/Learn\\_LaTeX\\_in\\_30\\_minutes](https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes)

- Should be useful for your future reports/articles/papers/theses (not only for this course!)
- Useful online tool, **Overleaf**: <https://www.overleaf.com/>

## Contacting us:

- Questions about assignments should be ideally asked during the dedicated sessions, but we also do answer questions by email (But you should not expect us to answer questions asked on the day of the deadline).
- **Important:** When you contact me, please **ALWAYS** put the two TAs in cc. (all email addresses are on page 1).

# More on the rules

## FAQ:

- Q: *Oh I've done my assignment in Keras, it's too late to change it to PyTorch. Can you still accept my solution, just for this time?*
- A: No, we only accept submission written in Python and PyTorch.

# More on the rules

## FAQ:

- Q: *Oh I've done my assignment in Keras, it's too late to change it to PyTorch. Can you still accept my solution, just for this time?*  
**A:** No, we only accept submission written in Python and PyTorch.
- Q: *I have too many assignments this week, can we postpone the deadline?*  
**A:** No. We publish the assignment (at least) two weeks in advance.

# More on the rules

## FAQ:

- Q: *Oh I've done my assignment in Keras, it's too late to change it to PyTorch. Can you still accept my solution, just for this time?*  
**A:** No, we only accept submission written in Python and PyTorch.
- Q: *I have too many assignments this week, can we postpone the deadline?*  
**A:** No. We publish the assignment (at least) two weeks in advance.
- Q: *Do you accept scan/picture of handwritten solution?*  
**A:** No, we only accept machine printed submissions.

# More on the rules

## FAQ:

- Q: *Oh I've done my assignment in Keras, it's too late to change it to PyTorch. Can you still accept my solution, just for this time?*  
**A:** No, we only accept submission written in Python and PyTorch.
- Q: *I have too many assignments this week, can we postpone the deadline?*  
**A:** No. We publish the assignment (at least) two weeks in advance.
- Q: *Do you accept scan/picture of handwritten solution?*  
**A:** No, we only accept machine printed submissions.
- Q: *Can you give me 2 more hours? ("I'm copying the solution from a friend")*  
**A:** No, we'll apply the late submission policy (Copying will result in 0 for both you and your friend).

# More on the rules

## FAQ:

- Q: *Oh I've done my assignment in Keras, it's too late to change it to PyTorch. Can you still accept my solution, just for this time?*  
**A:** No, we only accept submission written in Python and PyTorch.
- Q: *I have too many assignments this week, can we postpone the deadline?*  
**A:** No. We publish the assignment (at least) two weeks in advance.
- Q: *Do you accept scan/picture of handwritten solution?*  
**A:** No, we only accept machine printed submissions.
- Q: *Can you give me 2 more hours? ("I'm copying the solution from a friend")*  
**A:** No, we'll apply the late submission policy (Copying will result in 0 for both you and your friend).

## Other remarks:

- Please make use of our Q/A sessions to ask questions on assignments. We will not help you in the last minutes.  
Bad example: write to us a day before the deadline to complain about the assignment.

# 2022 Provisional Planning

Dates		10:30 - 11:15	≈ 11:15 - 12:00
September	19	Lecture (Intro)	Exercise 1
	26	Lecture (Sec. 1 + begin Sec. 2.1)	
October	03	Lecture (Sec. 2.1)	Exercise 2
	10	Lecture (Sec. 2.2)	Exercise 3 / <b>Assignment 1</b>
	17	Lecture (Sec. 3.1)	<b>Q/A Assign. 1 (cont'd)</b> & Exercise 4
	24	Exercise 5 / <b>Assignment 2</b>	
	31	Lecture (Sec. 3.2)	<b>Q/A Assign. 2 (cont'd)</b> & Exercise 6
November	7	Lecture (Sec. 3.3)	Exercise 6 (cont'd)
	14	Exercise 7 / <b>Assignment 3</b>	
	21	Lecture (Sec. 4)	<b>Q/A Assign. 3 (cont'd)</b>
	28	<b>No class</b> , Recorded Video: Exercise 8 / <b>Assignment 4</b>	
December	05		
	12	Lecture (Sec. 5 & 6)	<b>Q/A Assign. 4</b>
	19	<b>Q/A Assign. 4</b>	

# 2022 Provisional Planning (cont'd)

- The exact schedule from the end of November is still to be determined.
- We will make a poll to find the dates later (toward the end of October once the class is settled after 6 weeks).
- Anything after Nov. 28 in the previous slide is subject to change!

# Textbooks/ References/ Links

Many good materials are available **free online**.

For example:

- Textbooks on **machine learning** in general.

- Bishop (2006). Pattern Recognition and Machine Learning. <https://www.microsoft.com/en-us/research/people/cmbishop/prml-book/>
- Goodfellow et al. (2015) Deep Learning.  
<https://www.deeplearningbook.org/>
- Zhang et al. (2019). Dive into Deep Learning.  
<https://d2l.ai>
- Graves (2012).  
Supervised Sequence Labelling with Recurrent Neural Networks.  
<https://www.cs.toronto.edu/~graves/preprint.pdf>

# Textbooks/ References/ Links (cont'd)

## ■ Python

- <https://docs.python.org/3/tutorial/>  
**Highly recommended to check ASAP if not already familiar with Python!**
- Downey (2015). Think Python. 2nd Edition.  
<https://greenteapress.com/wp/think-python-2e/>
- Official quick intro to **NumPy**. **Same!**  
<https://numpy.org/devdocs/user/quickstart.html>

## ■ PyTorch

- Official tutorial: <https://pytorch.org/tutorials/>
- Deep Learning with PyTorch: A 60 minute blitz  
[https://pytorch.org/tutorials/beginner/deep\\_learning\\_60min\\_blitz.html](https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html)

**Exercises 1 and 2 will cover basics of Python/NumPy and PyTorch respectively.**

# Computational Resources

Your laptop is not a good option to run experiments. There are exercises you can not solve on your laptop.

# Computational Resources

Your laptop is not a good option to run experiments. There are exercises you can not solve on your laptop.

## ■ Google Colab <https://colab.research.google.com/notebooks/intro.ipynb>

- The default choice. Need to create a google account.
- Good tool for interactive programming (similar to Jupyter notebook).
- **Free GPUs** available (*Edit → Notebook Settings*, and choose GPU).
- In principle, all assignments are solvable using this resource. Warning: we only accept plain Python script. Please do not submit notebook file.

# Computational Resources

Your laptop is not a good option to run experiments. There are exercises you can not solve on your laptop.

## ■ **Google Colab** <https://colab.research.google.com/notebooks/intro.ipynb>

- The default choice. Need to create a google account.
- Good tool for interactive programming (similar to Jupyter notebook).
- **Free GPUs** available (*Edit → Notebook Settings*, and choose GPU).
- In principle, all assignments are solvable using this resource. Warning: we only accept plain Python script. Please do not submit notebook file.

## ■ **Kaggle** <https://www.kaggle.com/code>

- Backup option if you hit “some” utilization limit on Colab.
- Need to confirm a phone number to access free GPUs.
- *Settings* on the right panel → *Accelerator*, and choose GPU.

# Computational Resources

Your laptop is not a good option to run experiments. There are exercises you can not solve on your laptop.

## ■ **Google Colab** <https://colab.research.google.com/notebooks/intro.ipynb>

- The default choice. Need to create a google account.
- Good tool for interactive programming (similar to Jupyter notebook).
- **Free GPUs** available (*Edit → Notebook Settings*, and choose GPU).
- In principle, all assignments are solvable using this resource. Warning: we only accept plain Python script. Please do not submit notebook file.

## ■ **Kaggle** <https://www.kaggle.com/code>

- Backup option if you hit “some” utilization limit on Colab.
- Need to confirm a phone number to access free GPUs.
- *Settings* on the right panel → *Accelerator*, and choose GPU.

## ■ **USI, ICS cluster** <https://intranet.ics.usi.ch/HPC>

- Good option if you do not like writing code on a notebook.
- Learning how to organize your experiments on a cluster is good.
- You need to use/learn *Slurm*. Also risk: the cluster can be down!

**Please make sure that you can access Colab or Kaggle today.**

- General instructions: <https://intranet.ics.usi.ch/HPC>
- You will get credentials to access the cluster today ( `username` and `password` ).
- Connect to a front node `hpc.ics.usi.ch` (or 195.176.181.122) using SSH, i.e., execute on a terminal: `ssh username@hpc.usi.ch` (be careful with 0 vs. o vs. O when typing the password).
- Job manager used at ICS is *Slurm*

# USI ICS Cluster, Slurm

- You have to learn basic commands of *Slurm*, essentially:
  - `squeue` to view jobs in the queue.
  - `sbatch your_script_file.sh` to submit your job.
  - `scancel $JOB_ID` to kill your job.
- Correctly prepare `your_script_file.sh`. Example:

```
1 #!/bin/bash
2
3 #SBATCH --job-name="abc"
4 #SBATCH --output=abc.%j.out
5 #SBATCH --error=abc.%j.err
6 #SBATCH --partition=gpu
7 #SBATCH --time=00:15:00
8 #SBATCH --mem=4000
9 #SBATCH --exclusive
10
11 srun python my_code.py
```

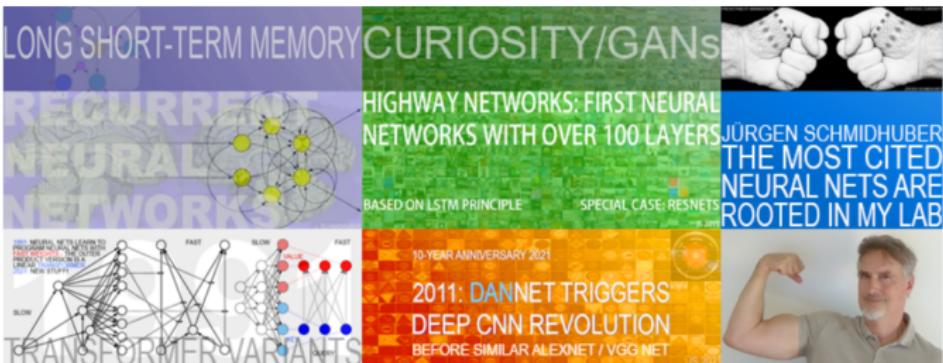
where `my_code.py` is your main Python code file.

- Check the *partitions* on <https://intranet.ics.usi.ch/HPC>

# Who are we?

We work in **Prof. Jürgen Schmidhuber's team** at

**IDSIA - Dalle Molle Institute for Artificial Intelligence**



- Lab with many historical contributions to deep learning and artificial intelligence  
<https://people.idsia.ch/~juergen/most-cited-neural-nets.html>.
- Affiliated with both USI and SUPSI.

# Your background?

- Undergraduate major?
- Programming experience?
- Python? C++?
- Machine Learning? Neural networks?
- Interest/application domain?

# Your background?

- Undergraduate major?
- Programming experience?
- Python? C++?
- Machine Learning? Neural networks?
- Interest/application domain?

This is a course for a broad audience.

But hopefully, you:

- have interests! e.g., some vision of applications you are interested in.
- are ready to spend time for hands-on learning, e.g., debugging your code.

# Deep learning: impact on a wide range of applications

## Image

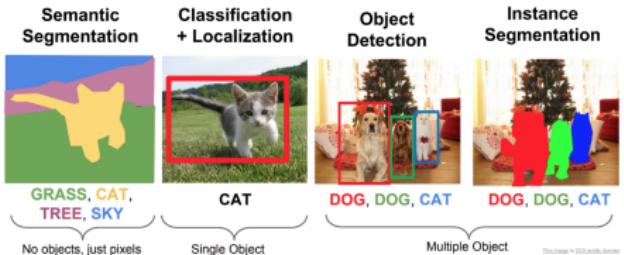
- Image classification
- Object detection/segmentation
- Image generation
- Image to image translation...



Monet → photo

Input: Image/Paiting

Output: Image/Photo-realistic picture



## ■ Image captioning

A white dog running along a beach



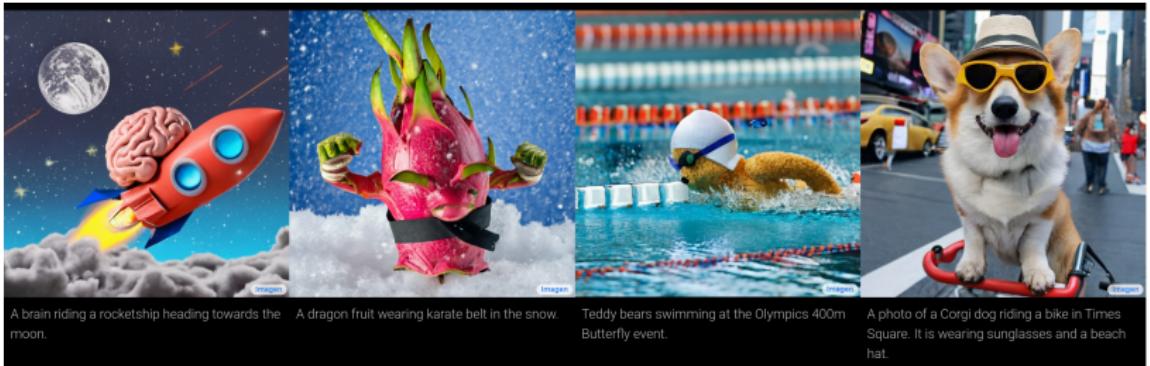
Input: Image

Output: Text describing the image

Images taken from [Nikolaus & Abdou<sup>+</sup> 19, Zhu & Park<sup>+</sup> 17, Li & Johnson<sup>+</sup> 17].

# Text to image (trending in 2022)

Imagen (Google, 2022) <https://imagen.research.google/>



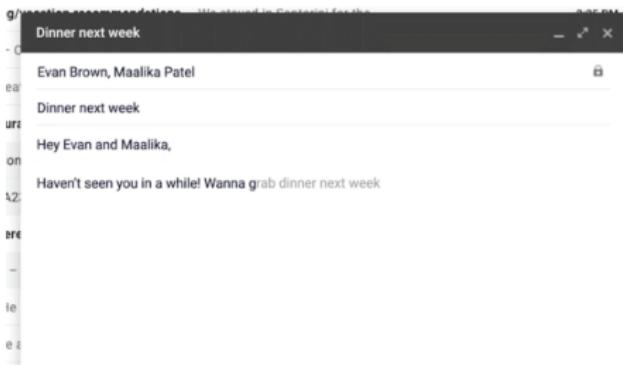
DALL-E 2 (OpenAI, 2022) <https://openai.com/dall-e-2/>



# Deep learning: impact on wide range of applications (cont'd)

## Natural language & speech

- Machine translation
- Automatic speech recognition
- Text to speech
- Text prediction/Language modeling
- Text classification, generation, summarization...
- Sentiment analysis
- Question answering



# Text generation (trending since 2019)

## GPT 2 & 3 Language models (OpenAI, 2019/2020)

Q: What is your favorite animal?

A: My favorite animal is a dog.

Q: Why?

A: Because dogs are loyal and friendly.

Q: What are two reasons that a dog might be in a bad mood?

A: Two reasons that a dog might be in a bad mood are if it is hungry or if it is hot.

Q: How many eyes does a giraffe have?

A: A giraffe has two eyes.

example taken from

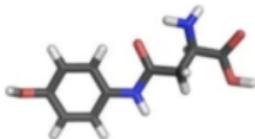
<https://lacker.io/ai/2020/07/06/giving-gpt-3-a-turing-test.html>

Similar model applied to code generation: OpenAI Codex (2021)

Demo: <https://www.youtube.com/watch?v=SGUCcjHTmGY>

# Deep learning applications: many more domains...

- Game playing (board/video games)
- Robotics
- Art and Music (generation)
- Medical/pharmaceutical domain, drug discovery, chemical reaction prediction...
- Financial engineering...



Figures taken from [Finn & Yu<sup>+</sup> 17, Kim & Kang<sup>+</sup> 18, Mnih & Kavukcuoglu<sup>+</sup> 13]

# In this lecture...

- Unfortunately, we do not have enough time to cover everything...
- But we'll study basic but fundamental tasks to illustrate various things that we can do with neural networks (the focus is on supervised learning).
- Fundamental ideas are transferable to other tasks beyond in this course, and prepare you for future learning.
- Application domains covered in the exercises:
  - **Image classification** with feed-forward and **convolutional neural networks**.
  - **Text generation** (and related tasks) with **recurrent neural networks**.
  - **Mathematical problem solving** (and/or related translation-like tasks) with **Transformers**.

- 1. Workflow Overview**
- 2. Introduction to Tools**
  1. Basic Tools
  2. Implementation of Systems
- 3. Fundamental building blocks**
  1. Feed-forward NNs, Convolutional NNs, Residual NNs
  2. Recurrent NNs and LSTM
  3. Attention, Self-attention, Transformers
- 4. Building models**
- 5. Practical tricks and methods for training**
- 6. Final words and Outlook**

- 1. Workflow Overview**
2. Introduction to Tools
3. Fundamental building blocks
4. Building models
5. Practical tricks and methods for training
6. Final words and Outlook

# Introduction

- High-level principle:
  - data: many input/output example pairs.
  - models **learn from examples**.

# Introduction

- High-level principle:
  - data: many input/output example pairs.
  - models **learn from examples**.
- For example:
  - You will **not** write a hard-coded program which translates a English sentence to a French sentence.

# Introduction

- High-level principle:
  - data: many input/output example pairs.
  - models **learn from examples**.
- For example:
  - You will **not** write a hard-coded program which translates a English sentence to a French sentence.
  - Instead: you will **train** your **model/neural network** by letting it see many (many!) examples of paired English-French sentences (**training data**).

# Introduction

- High-level principle:
  - data: many input/output example pairs.
  - models **learn from examples**.
- For example:
  - You will **not** write a hard-coded program which translates a English sentence to a French sentence.
  - Instead: you will **train** your **model/neural network** by letting it see many (many!) examples of paired English-French sentences (**training data**).
- Your model has **parameters**, which are initially random.
  - **Training will modify/optimize the model parameters** such that they would minimize a **loss function** (correlated to the task you want to solve).

# Introduction

- High-level principle:
  - data: many input/output example pairs.
  - models **learn from examples**.
- For example:
  - You will **not** write a hard-coded program which translates a English sentence to a French sentence.
  - Instead: you will **train** your **model/neural network** by letting it see many (many!) examples of paired English-French sentences (**training data**).
- Your model has **parameters**, which are initially random.
  - **Training will modify/optimize the model parameters** such that they would minimize a **loss function** (correlated to the task you want to solve).
- Your model and training setup have **hyper-parameters**:
  - You train multiple models with different hyper-parameters and select the best model based on its **validation data** performance.

# Introduction

- High-level principle:
  - data: many input/output example pairs.
  - models **learn from examples**.
- For example:
  - You will **not** write a hard-coded program which translates a English sentence to a French sentence.
  - Instead: you will **train** your **model/neural network** by letting it see many (many!) examples of paired English-French sentences (**training data**).
- Your model has **parameters**, which are initially random.
  - **Training will modify/optimize the model parameters** such that they would minimize a **loss function** (correlated to the task you want to solve).
- Your model and training setup have **hyper-parameters**:
  - You train multiple models with different hyper-parameters and select the best model based on its **validation data** performance.
- The final model is evaluated on the **test data**.

# Typical Deep Learning Workflow (supervised learning)

- Define/obtain: **task, dataset**
- Define **model**
- Define **loss** and **optimization algorithm**
- Do **training** and **hyper-parameter tuning**
- Evaluate its performance using an **evaluation metric(s)**  
(:= make prediction on the test data using the trained model; requires some search algorithm depending on the task)

Image classification? Speech recognition? Image question answering?

For any problem, you should know your **task** and **dataset**:

- What are the **inputs & outputs** to the system?
- What are the **basic statistics** of the data?  
(size of the data, number of output classes, ...)
- Which metric do you use to **evaluate** the final model performance?

General recommendation: take some time looking into data examples, before building a model.

# Task and Data (cont'd)

## Example 1:

- Task: **Image classification**

# Task and Data (cont'd)

## Example 1:

- Task: **Image classification**
- Nature of the problem: input = image, output = class label.

# Task and Data (cont'd)

## Example 1:

- Task: **Image classification**
- Nature of the problem: input = image, output = class label.
- Dataset statistics? For example:
  - Number of training samples: 10K 100K? 1M? images (is this big? toy task?)
  - Number of classes: 10? 100? 1000? classes (dog, cat, airplane,...)  
(is this big? small/toy task?)

# Task and Data (cont'd)

## Example 1:

- Task: **Image classification**
- Nature of the problem: input = image, output = class label.
- Dataset statistics? For example:
  - Number of training samples: 10K 100K? 1M? images (is this big? toy task?)
  - Number of classes: 10? 100? 1000? classes (dog, cat, airplane,...)  
(is this big? small/toy task?)
- Evaluation metric: classification error or accuracy (number of correctly classified test images divided by the total number of test images).
- Extra aspect: is this a standard benchmark dataset? do people publish on this dataset? What are the baseline models/performance?

# Task and Data (cont'd)

## Example 2:

- Task: Machine Translation

# Task and Data (cont'd)

## Example 2:

- Task: **Machine Translation**

- Nature of the problem:

input = text in the source language, output = text in the target language.  
Each example: bilingual sentence pair.

# Task and Data (cont'd)

## Example 2:

- Task: **Machine Translation**
- Nature of the problem:  
input = text in the source language, output = text in the target language.  
Each example: bilingual sentence pair.
- Dataset statistics? For example:
  - Number of training sample:  
100 M bilingual training sentence pairs (is this big? small/toy task?)
  - Vocabulary: word? sub-word? character? (choice for your system)
  - Vocabulary size: 30 K in source and 40 K tokens in target language.
  - (Domain of texts: News? European parliament?)
- Evaluation measure: BLEU [\[Papineni & Roukos<sup>+</sup> 02\]](#).

- Architecture & type of your model will depend on the task.
- In particular: different input/output modalities.
- **In Section 3:** different types of neural network architectures will be reviewed.
- Here: quick reminder on basics of neural networks, to introduce **key words** and practicalities.

# Modeling: Reminders on neural networks

A Neural Network:

- is a **parameterized** function (whose parameters are *learned* from data).

# Modeling: Reminders on neural networks

A Neural Network:

- is a **parameterized** function (whose parameters are *learned* from data).
- transforms a **vector to another vector** (the most standard case).

# Modeling: Reminders on neural networks

A Neural Network:

- is a **parameterized** function (whose parameters are *learned* from data).
- transforms a **vector to another vector** (the most standard case).
- Such a layer can be composed multiple times (output of a layer becomes input to next layer...etc) to make the model **deeper**.  
Each layer with its own parameters (in principle).

# Modeling:

## Reminders on neural networks

**Two basic operations.** Let  $d_{\text{in}}$  and  $d_{\text{out}}$  denote positive integers.

A typical **one-layer** feed-forward neural network transforms input  $x \in \mathbb{R}^{d_{\text{in}}}$  to output  $z \in \mathbb{R}^{d_{\text{out}}}$  as follows:

- **Affine transformation (Linear layer):**  $y = Wx + b$  where  $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$  is a **weight matrix** and  $b \in \mathbb{R}^{d_{\text{out}}}$  is a **bias** vector.  $y \in \mathbb{R}^{d_{\text{out}}}$ .  
 $W$  and  $b$  are **trainable parameters**.

$$\begin{matrix} \mathbf{y} \in \mathbb{R}^{d_{\text{out}}} \\ \begin{array}{|c|c|c|c|} \hline & \text{white circle} & & \\ \hline & \text{white circle} & & \\ \hline & \text{white circle} & & \\ \hline \end{array} \end{matrix} = \begin{matrix} \mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline \end{array} \end{matrix} \times \begin{matrix} \mathbf{x} \in \mathbb{R}^{d_{\text{in}}} \\ \begin{array}{|c|c|c|} \hline & \text{white circle} & \\ \hline \end{array} \end{matrix}$$

# Modeling:

## Reminders on neural networks

**Two basic operations.** Let  $d_{\text{in}}$  and  $d_{\text{out}}$  denote positive integers.

A typical **one-layer** feed-forward neural network transforms input  $x \in \mathbb{R}^{d_{\text{in}}}$  to output  $z \in \mathbb{R}^{d_{\text{out}}}$  as follows:

- **Affine transformation (Linear layer):**  $y = Wx + b$  where  $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$  is a **weight matrix** and  $b \in \mathbb{R}^{d_{\text{out}}}$  is a **bias** vector.  $y \in \mathbb{R}^{d_{\text{out}}}$ .  
 $W$  and  $b$  are **trainable parameters**.

$$\begin{matrix} \mathbf{y} \in \mathbb{R}^{d_{\text{out}}} \\ \begin{array}{|c|c|c|c|} \hline & \text{white circle} & & \\ \hline & \text{white circle} & & \\ \hline & \text{white circle} & & \\ \hline \end{array} \end{matrix} = \begin{matrix} \mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline \end{array} \end{matrix} \times \begin{matrix} \mathbf{x} \in \mathbb{R}^{d_{\text{in}}} \\ \begin{array}{|c|c|c|} \hline & \text{green circle} & \\ \hline \end{array} \end{matrix}$$

- **Element-wise non-linear activation function:**  $z = \sigma(y) \in \mathbb{R}^{d_{\text{out}}}$

# Modeling:

## Reminders on neural networks

**Two basic operations.** Let  $d_{\text{in}}$  and  $d_{\text{out}}$  denote positive integers.

A typical **one-layer** feed-forward neural network transforms input  $x \in \mathbb{R}^{d_{\text{in}}}$  to output  $z \in \mathbb{R}^{d_{\text{out}}}$  as follows:

- **Affine transformation (Linear layer):**  $y = Wx + b$  where  $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$  is a **weight matrix** and  $b \in \mathbb{R}^{d_{\text{out}}}$  is a **bias** vector.  $y \in \mathbb{R}^{d_{\text{out}}}$ .  
 $W$  and  $b$  are **trainable parameters**.

$$\begin{matrix} \mathbf{y} \in \mathbb{R}^{d_{\text{out}}} \\ \begin{array}{|c|c|c|c|} \hline & \text{orange circle} & \text{orange circle} & \text{orange circle} \\ \hline & \text{orange circle} & \text{orange circle} & \text{orange circle} \\ \hline & \text{orange circle} & \text{orange circle} & \text{orange circle} \\ \hline \end{array} \end{matrix} = \begin{matrix} \mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}} \\ \begin{array}{|c|c|c|c|} \hline & \text{blue square} & \text{blue square} & \text{blue square} \\ \hline & \text{blue square} & \text{blue square} & \text{blue square} \\ \hline & \text{blue square} & \text{blue square} & \text{blue square} \\ \hline & \text{blue square} & \text{blue square} & \text{blue square} \\ \hline \end{array} \end{matrix} \times \begin{matrix} \mathbf{x} \in \mathbb{R}^{d_{\text{in}}} \\ \begin{array}{|c|c|c|} \hline & \text{green circle} & \text{green circle} \\ \hline & \text{green circle} & \text{green circle} \\ \hline & \text{green circle} & \text{green circle} \\ \hline \end{array} \end{matrix}$$

- Element-wise non-linear **activation function**:  $z = \sigma(y) \in \mathbb{R}^{d_{\text{out}}}$

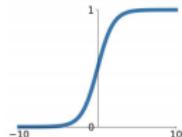
We can compose many layers (an output of a layer becomes an input to the next layer...etc) to obtain a **deep** neural net.

# Modeling: Reminders on neural networks (cont'd)

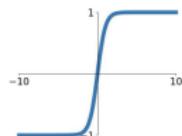
## Activation functions.

- Commonly used functions:  
**sigmoid**, **tanh**,  
rectifier linear unit (**ReLU**).
- More exotic functions:  
ELU, GELU, SiLU...
- For normalized output: **softmax**.

**Sigmoid**

$$\sigma(x) = \frac{1}{1+e^{-x}}$$


**tanh**

$$\tanh(x)$$


**ReLU**

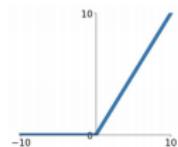
$$\max(0, x)$$


Figure taken from [Li & Johnson<sup>+</sup> 19b]

# Activation functions (cont'd)

## Softmax function:

- Let  $d$  denote a positive integer
- Input  $x \in \mathbb{R}^d$  is an arbitrary vector.
- Output  $y \in \mathbb{R}^d$  defines a **probability distribution**

For  $1 \leq i \leq d$ ,  $i$ -th entry of vector  $y \in \mathbb{R}^d$  is defined as:

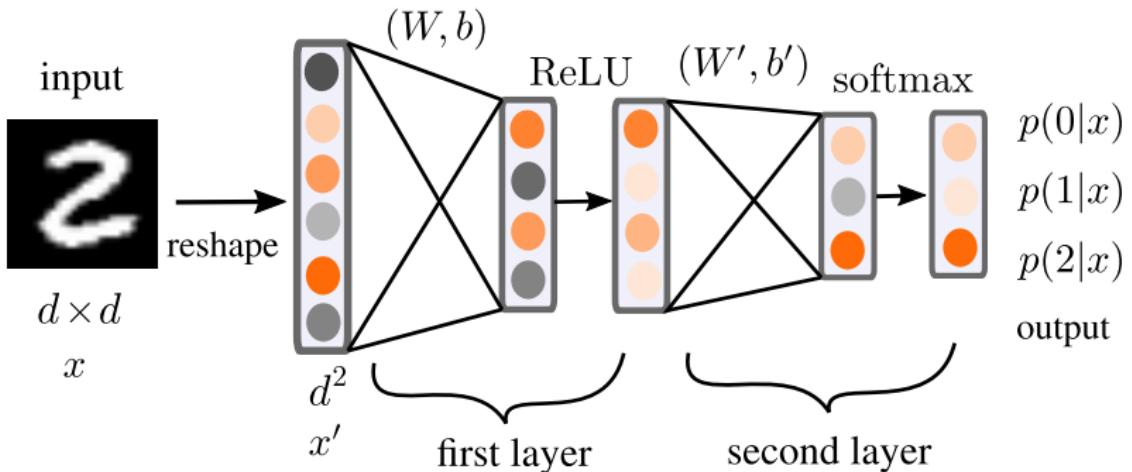
$$y_i = \frac{\exp(x_i)}{\sum_{n=1}^d \exp(x_n)}$$

The output vector  $y$  has the following properties:

- All entries which are positive: For  $1 \leq i \leq d$ ,  $y_i > 0$ .
- The sum of all entries is one (normalized output):  $\sum_{i=1}^d y_i = 1$ .

The input  $x$  to the softmax function is often called **logit**.

# Model: illustration



- 2-layer feed-forward neural network model for classification of  $d$  by  $d$  images of digits to 3 possible classes: 0, 1 or 2.

**Training:** optimization process to find the optimal values for the model parameters for the loss function.

- Before training: model parameters are random.
- After training: model parameters should have values which allow the model to solve the task!

For that we must define:

- **Loss function:** function of model parameters to be minimized during training.
- **Optimizer:** specification of optimization algorithm with its hyper-parameters.

**Training hyper-parameters** need to be tuned: e.g. learning rate...

# Training: Loss

- Let  $N$  and  $P$  denote positive integers.  
A super script  $i$  in  $x^i$  denotes  $i$ -th training example.
- **Training data**  $\{(x^1, y^1), \dots, (x^N, y^N)\}$   
is a set of  $N$  input  $x^i$  and output/target  $y^i$  pairs.
- The exact specification of the nature of  $x^i$  and  $y^i$  does not matter here (you can assume that they are both vectors).

# Training: Loss

- Let  $N$  and  $P$  denote positive integers.  
A super script  $i$  in  $x^i$  denotes  $i$ -th training example.
- **Training data**  $\{(x^1, y^1), \dots, (x^N, y^N)\}$   
is a set of  $N$  input  $x^i$  and output/target  $y^i$  pairs.
- The exact specification of the nature of  $x^i$  and  $y^i$  does not matter here (you can assume that they are both vectors).
- We want to train our **model**  $f_\theta$  with  $P$  trainable **parameters**  $\theta \in \mathbb{R}^P$ .

- **Loss function**  $\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N \ell(f_\theta(x^i), y^i)$

with some function  $\ell$  (dependent of the problem) which “*compares*” each model output/prediction  $f_\theta(x^i)$  and the target (true label)  $y^i$ .

# Training: Loss (cont'd), Reminders

For **regression** problems, let  $d$  denote a positive integer (target vector size)

- Target  $y^i \in \mathbb{R}^d$  is a **vector**
- so is the model output  $f_\theta(x^i) \in \mathbb{R}^d$ .

Using the squared error  $\ell(f_\theta(x^i), y^i) = \|f_\theta(x^i) - y^i\|_2^2$  we get the

**Mean Squared Error (MSE) Loss:**  $\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N \|f_\theta(x^i) - y^i\|_2^2$

# Training: Loss (cont'd), Reminders

For **classification** problems, let  $C$  denote a positive integer (number of output classes).

- Target  $y^i \in \{1 \dots C\}$  is a **label**/integer
- Model output  $f_\theta(x^i) \in \mathbb{R}^C$  is a vector defining a **probability over the class labels**:

$$f_\theta(x^i) = [p_\theta(1|x^i), \dots, p_\theta(k|x^i), \dots, p_\theta(C|x^i)] \text{ for } k \in \{1 \dots C\}$$

where  $p_\theta(k|x^i) \in \mathbb{R}$  is a probability that the input is of class  $k$  according to the model.

Using cross entropy (where  $\delta_{y^i, k} = 1$  if  $y^i = k$ , 0 otherwise)

$$\ell(f_\theta(x^i), y^i) = - \sum_{k=1}^C \delta_{y^i, k} \log p_\theta(k|x^i) = - \log p_\theta(y^i|x^i) \quad \text{we get the}$$

**Cross-Entropy Loss:**  $\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^N \log p_\theta(y^i|x^i)$

# Training: Optimization/mini-batch

Having defined the loss function  $\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N \ell(f_\theta(x^i), y^i)$

## Gradient descent:

- is an iterative process. At iteration step  $n$ , update  $\theta(n) \in \mathbb{R}^P$  to  $\theta(n+1) \in \mathbb{R}^P$  by
- computating gradients  $\nabla_{\theta} \mathcal{L}(\theta(n)) \in \mathbb{R}^P$  (e.g., by **backpropagation**).
- applying one gradient descent step:  $\theta(n+1) = \theta(n) - \alpha * \nabla_{\theta} \mathcal{L}(\theta(n))$  where  $\alpha \in \mathbb{R}_+$  is **learning rate** (hyper-parameter!)

# Training: Optimization/mini-batch

Having defined the loss function  $\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N \ell(f_\theta(x^i), y^i)$

## Gradient descent:

- is an iterative process. At iteration step  $n$ , update  $\theta(n) \in \mathbb{R}^P$  to  $\theta(n+1) \in \mathbb{R}^P$  by
  - computating gradients  $\nabla_{\theta} \mathcal{L}(\theta(n)) \in \mathbb{R}^P$  (e.g., by **backpropagation**).
  - applying one gradient descent step:  $\theta(n+1) = \theta(n) - \alpha * \nabla_{\theta} \mathcal{L}(\theta(n))$  where  $\alpha \in \mathbb{R}_+$  is **learning rate** (hyper-parameter!)
- This is repeated multiple times (needs to define some stopping criterion)

# Training: Optimization/mini-batch

Having defined the loss function  $\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N \ell(f_\theta(x^i), y^i)$

## Gradient descent:

- is an iterative process. At iteration step  $n$ , update  $\theta(n) \in \mathbb{R}^P$  to  $\theta(n + 1) \in \mathbb{R}^P$  by
- computating gradients  $\nabla_{\theta} \mathcal{L}(\theta(n)) \in \mathbb{R}^P$  (e.g., by **backpropagation**).
- applying one gradient descent step:  $\theta(n + 1) = \theta(n) - \alpha * \nabla_{\theta} \mathcal{L}(\theta(n))$  where  $\alpha \in \mathbb{R}_+$  is **learning rate** (hyper-parameter!)
- This is repeated multiple times (needs to define some stopping criterion)
- **Gradients computed on the whole dataset (average over  $N$ ) for each update. Inefficient?**

# Training: Optimization/mini-batch

Having defined the loss function  $\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N \ell(f_\theta(x^i), y^i)$

## Gradient descent:

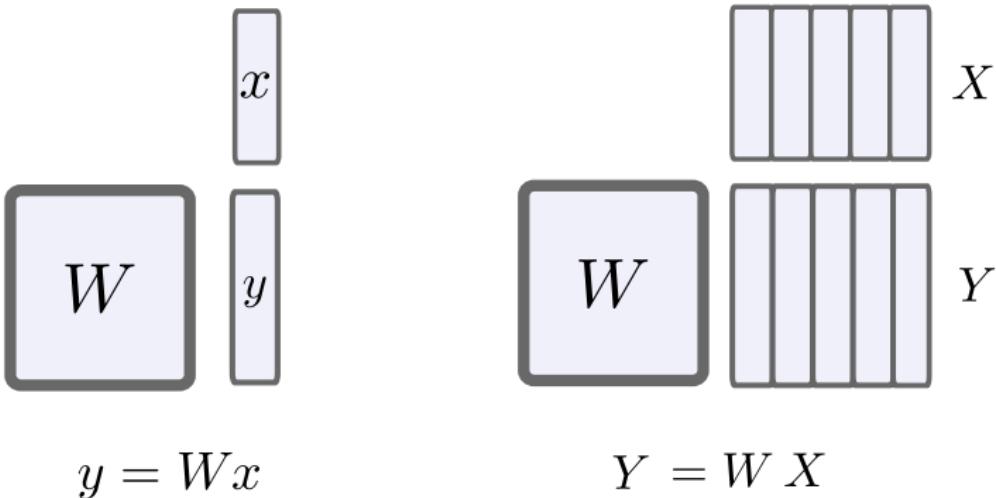
- is an iterative process. At iteration step  $n$ , update  $\theta(n) \in \mathbb{R}^P$  to  $\theta(n+1) \in \mathbb{R}^P$  by
- computating gradients  $\nabla_{\theta} \mathcal{L}(\theta(n)) \in \mathbb{R}^P$  (e.g., by **backpropagation**).
- applying one gradient descent step:  $\theta(n+1) = \theta(n) - \alpha * \nabla_{\theta} \mathcal{L}(\theta(n))$  where  $\alpha \in \mathbb{R}_+$  is **learning rate** (hyper-parameter!)
- This is repeated multiple times (needs to define some stopping criterion)
- **Gradients computed on the whole dataset (average over  $N$ ) for each update. Inefficient?**

## Alternative: update parameters using gradients computed on mini-batches

- Gradients computed on a few, randomly selected data points (mini-batch or simply *batch*)
- Number of data points in a batch is the **batch size** (also hyper-parameter!)
- Allows more frequent updates (works well in practice)

## Note: parallel computation

e.g. processing  $N$  inputs to a linear layer



$W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ ,  $y \in \mathbb{R}^{d_{\text{out}}}$  and  $x \in \mathbb{R}^{d_{\text{in}}}$  **VS.**  $Y \in \mathbb{R}^{d_{\text{out}} \times N}$  and  $X \in \mathbb{R}^{d_{\text{in}} \times N}$

- Computing linear transformation of **multiple examples** can be packed into a **single** matrix-matrix multiplication.
- Benefit from efficient parallel computation (especially on GPUs).
- Should write **vectorized code** instead of loop (example later)

# Training: Optimization/Optimizer

- The parameter update equation in the previous slide:

$$\theta(n+1) = \theta(n) - \alpha * \nabla_{\theta} \mathcal{L}(\theta(n))$$

with gradients computed on a mini-batch corresponds to **stochastic gradient descent (SGD)** algorithm.

# Training: Optimization/Optimizer

- The parameter update equation in the previous slide:  
$$\theta(n + 1) = \theta(n) - \alpha * \nabla_{\theta} \mathcal{L}(\theta(n))$$
with gradients computed on a mini-batch corresponds to **stochastic gradient descent (SGD)** algorithm.
- This makes use of the same learning rate  $\alpha$  for all parameters.
  - Many other variants for optimization algorithm with adaptive learning rates for each parameter has been proposed. Popular overview:  
<https://ruder.io/optimizing-gradient-descent/>

# Training: Optimization/Optimizer

- The parameter update equation in the previous slide:

$$\theta(n+1) = \theta(n) - \alpha * \nabla_{\theta} \mathcal{L}(\theta(n))$$

with gradients computed on a mini-batch corresponds to **stochastic gradient descent (SGD)** algorithm.

- This makes use of the same learning rate  $\alpha$  for all parameters.
  - Many other variants for optimization algorithm with adaptive learning rates for each parameter has been proposed. Popular overview:  
<https://ruder.io/optimizing-gradient-descent/>
  - *adaptive*: change the effective learning rate depending on the past gradients of the corresponding parameter.
    - In particular, *Adam* [Kingma & Ba 15] is very popular.

# Training: Optimization/Optimizer

- The parameter update equation in the previous slide:

$$\theta(n+1) = \theta(n) - \alpha * \nabla_{\theta} \mathcal{L}(\theta(n))$$

with gradients computed on a mini-batch corresponds to **stochastic gradient descent (SGD)** algorithm.

- This makes use of the same learning rate  $\alpha$  for all parameters.
  - Many other variants for optimization algorithm with adaptive learning rates for each parameter has been proposed. Popular overview:  
<https://ruder.io/optimizing-gradient-descent/>
- *adaptive*: change the effective learning rate depending on the past gradients of the corresponding parameter.
  - In particular, *Adam* [Kingma & Ba 15] is very popular.

- **Choosing an optimizer = specifying the choice of optimization algorithm and its hyper-parameters.**

- Hands-on experience in exercises.

# Some Training Jargon

**Progress of training** is often described in terms of number of:

- **Epochs**: 1 epoch = one run over the whole training data.
- **Steps**: 1 step = 1 gradient update (typically using 1 mini-batch).
- **Updates**: normally synonym to steps.
- More rarely, **sub-epochs**: some fixed fraction of epoch,  
e.g. 1/4 of epoch.

The descriptions above are typically correct.

But unconventional definitions can also be found in some papers...

# Hyper-Parameter Tuning

Models have **hyper-parameters** such as:

- Number of layers, size of each hidden layer, other model specific hyper-parameters...

# Hyper-Parameter Tuning

Models have **hyper-parameters** such as:

- Number of layers, size of each hidden layer, other model specific hyper-parameters...

So do training algorithms:

- Learning rate, batch size, other optimizer specific hyper-parameters...

# Hyper-Parameter Tuning

Models have **hyper-parameters** such as:

- Number of layers, size of each hidden layer, other model specific hyper-parameters...

So do training algorithms:

- Learning rate, batch size, other optimizer specific hyper-parameters...

Hyper-parameters are not *learned* by the training algorithm!

- **Tuning**: specify a set of hyper-parameters, train the model, and check its performance on validation data, repeat until you obtain a "good" model.
- Important, because: **model performance can highly depend on the hyper-parameter tuning!**
- Tuning is a practice you learn from hands-on experience.

# Summary

## What have we learned?

- Overview of workflow: main concepts, key words.
  - task & data, nature of the problem, statistics.
  - evaluation measure
  - model
  - training
  - loss function
  - stochastic gradient descent
  - hyper-parameter tuning

## Coming up next...

- How can we implement these concepts concretely?
- Introduction to tools for that.
- ...

# Outline

1. Workflow Overview
- 2. Introduction to Tools**
  1. Basic Tools
  2. Implementation of Systems
3. Fundamental building blocks
4. Building models
5. Practical tricks and methods for training
6. Final words and Outlook

# Outline

1. Workflow Overview
2. **Introduction to Tools**
  1. Basic Tools
  2. Implementation of Systems
3. Fundamental building blocks
4. Building models
5. Practical tricks and methods for training
6. Final words and Outlook

- **Practical implementation** implies a number of **choices** in terms of framework or software to use. For example:
- Programming language: Python
- Deep learning library: PyTorch
- and more...
- This section: introduction to these practical *tools*.

- Again: if you are not familiar with Python, check:

<https://docs.python.org/3/tutorial/>

- High-level, general purpose, programming language

- "easy to use", "broadly popular"
- The most popular language for deep learning today
- Many libraries are available.
- A few characteristics:  
*indentation* is part of syntax, dynamically typed, indices start from 0... etc

- **Version** to be used: Python 3.6+.

- You will still find Python 2.x code...

But for any new code use: Python 3.6+.

# Python, simple illustration

```
1 def bubble_sort(list_nums):  
2     # Implement Bubble Sort.  
3     already_sorted = True  
4     numbers = list_nums.copy()  
5     n = len(numbers)  
6     for i in range(n):  
7         for j in range(n-i-1):  
8             if numbers[j] > numbers[j+1]:  
9                 numbers[j], numbers[j+1] = numbers[j+1], numbers[j]  
10                already_sorted = False  
11            if already_sorted:  
12                break  
13        return numbers
```

# Python, simple illustration

```
1 def bubble_sort(list_nums):  
2     # Implement Bubble Sort.  
3     already_sorted = True  
4     numbers = list_nums.copy()  
5     n = len(numbers)  
6     for i in range(n):  
7         for j in range(n-i-1):  
8             if numbers[j] > numbers[j+1]:  
9                 numbers[j], numbers[j+1] = numbers[j+1], numbers[j]  
10                already_sorted = False  
11            if already_sorted:  
12                break  
13        return numbers
```

```
1 >>> my_numbers = [58, 37, 2, 11, 96, 15, 64]  
2 >>> bubble_sort(my_numbers)  
3 [2, 11, 15, 37, 58, 64, 96]
```

# Python, simple illustration

```
1 def bubble_sort(list_nums):  
2     # Implement Bubble Sort.  
3     already_sorted = True  
4     numbers = list_nums.copy()  
5     n = len(numbers)  
6     for i in range(n):  
7         for j in range(n-i-1):  
8             if numbers[j] > numbers[j+1]:  
9                 numbers[j], numbers[j+1] = numbers[j+1], numbers[j]  
10                already_sorted = False  
11            if already_sorted:  
12                break  
13        return numbers
```

```
1 >>> my_numbers = [58, 37, 2, 11, 96, 15, 64]  
2 >>> bubble_sort(my_numbers)  
3 [2, 11, 15, 37, 58, 64, 96]
```

Though you should use **built-in** function instead (more efficient).

```
1 >>> sorted(my_numbers)  
2 [2, 11, 15, 37, 58, 64, 96]
```

## Python, simple illustration 2

Loops in Python are very simple (check Sec. 5.6 in the tutorial)

```
1 # dictionary storing count of fruits.  
2 >>> basket = {'apple': 4, 'banana': 2, 'orange': 5}  
3 >>> for fruit, count in basket.items():  
4 ...     print(fruit, count)  
5 ...  
6 apple 4  
7 banana 2  
8 orange 5
```

# Python, simple illustration 2

Loops in Python are very simple (check Sec. 5.6 in the tutorial)

```
1 # dictionary storing count of fruits.  
2 >>> basket = {'apple': 4, 'banana': 2, 'orange': 5}  
3 >>> for fruit, count in basket.items():  
4 ...     print(fruit, count)  
5 ...  
6 apple 4  
7 banana 2  
8 orange 5
```

```
1 # loop over a list, with loop counter:  
2 >>> for i, v in enumerate(['tic', 'tac', 'toe']):  
3 ...     print(i, v)  
4 ...  
5 0 tic  
6 1 tac  
7 2 toe
```

# Python, many packages

Different packages provide you with useful functions.

E.g. `math` module for quick calculations: (e.g. you want to immediately know the log of some number!)

```
1 >>> import math
2 >>> math.log(0.1)
3 -2.3025850929940455
4 >>> y = math.log(0.1)
5 >>> z = math.exp(y)
6 >>> z
7 0.1000000000000002
```

- **NumPy** `numpy`: manipulation of multi-dimensional arrays.
  - Seen in [Exercise 1](#) !
- **Matplotlib** `matplotlib`: tool for visualization, generate plots.
  - More in [Exercise 2](#) !
- etc...

The basic **package installer** for Python is pip (or pip3 or Python 3).

- To install the latest version of “SomeProject”:

```
1 pip3 install "SomeProject"
```

- To install a specific version (e.g. version 1.1):

```
1 pip3 install "SomeProject==1.1"
```

# Code Style

**Need guidelines to write consistent (easy to read) code.**

- How should I name my function? my variable? class?
- How should I space? indent? comment?
- What is the longest acceptable line length?
- How to split long lines... etc!

**Some reference guidelines:**

- PEP 8 <https://www.python.org/dev/peps/pep-0008/>
- Google guideline  
<https://github.com/google/styleguide/blob/gh-pages/pyguide.md>

**Crucial importance for collaborative projects!**

- When you ask other people to **review** your code.
- But also for yourself (to ease reading your own code).
- **Consistent** formatting makes reading much easier.

## Up to you!

- Emacs, vim, nano, kate, VSCode, ...
- If you do not know, you can try **VSCode** or **PyCharm** (Community).
  - Very convenient for both reading (navigating through a large project) and writing (with less formatting errors).
- What I use: vim (for simple edits) and VSCode in general.

# Managing environment

(Excursion; no need to worry about this in this course)

- Python, libraries, softwares, ... evolve over time.
- Resulting in code with **different versions** for each tool.
  - Currently working code has no guarantee to work again in another environment.
  - Also: often you will be using code that you find on the internet (Github), that someone had written, you do not know when, which requires some specific sets of versions for libraries...

**Environment managing tools** can help handling multiple configurations.

- Typical tools: `virtualenv`, `conda`
- If you do not have any preference yet, try `Miniconda`.
  - <https://docs.conda.io/en/latest/miniconda.html>
  - <https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html>

# Other useful tools

- Internet and Google!
- Very likely, the same problem you encountered has been already solved by other people!
- Search your problem, and you might find solutions on e.g. *StackOverflow*.
- Be careful with version mismatch.

# Libraries for Deep Learning

**Many possibilities, different teams/companies.**

- TensorFlow (Google), MXnet (Amazon), JAX (Google), ...
- Other high-level framework: keras (Google), ...

# Libraries for Deep Learning

**Many possibilities, different teams/companies.**

- TensorFlow (Google), MXnet (Amazon), JAX (Google), ...
- Other high-level framework: keras (Google), ...

**Side notes:**

- Theano (UdeM): end of support in 2017.
- Chainer (Preferred Networks): migration to PyTorch in 2019.
- Caffe (Berkeley, Facebook) merged to PyTorch in 2018.
- Cognitive Toolkit (Microsoft): 2016-2017.
- even older... Quicknet (Berkeley)

# Libraries for Deep Learning

**Many possibilities, different teams/companies.**

- TensorFlow (Google), MXnet (Amazon), JAX (Google), ...
- Other high-level framework: keras (Google), ...

**Side notes:**

- Theano (UdeM): end of support in 2017.
- Chainer (Preferred Networks): migration to PyTorch in 2019.
- Caffe (Berkeley, Facebook) merged to PyTorch in 2018.
- Cognitive Toolkit (Microsoft): 2016-2017.
- even older... Quicknet (Berkeley)

This lecture: **PyTorch**

- Development led by Meta/Facebook AI Research.
- Last week: became a project under Linux Foundation
- Very popular already. Increasing popularity...
- Used for example by OpenAI, Tesla...

# Introduction to PyTorch

## Two core aspects:

- GPU-friendly tensors.
- Automatic differentiation.

Note: current version PyTorch 1.12.1 (Aug. 2022) ~~PyTorch 1.9.0 (June 2021)~~  
~~PyTorch 1.6.0 (July 2020)~~

# Introduction to PyTorch (cont'd)

## Packages which you will be often importing:

- `torch` : top-level package.
- `torch.nn` : modules for building neural networks.
- `torch.nn.functional` : various functions.
- `torch.optim` : optimization related tools.
- `torch.utils` : handling data etc.
- ...

Many examples later.

# Introduction to PyTorch (cont'd)

**Many packages are available.**

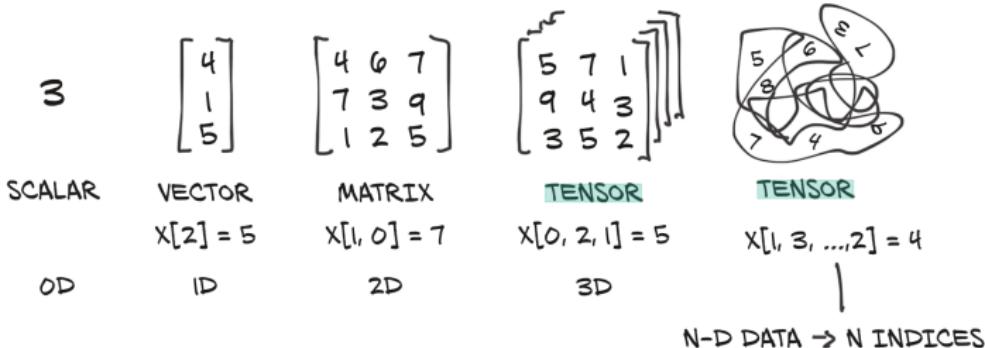
E.g. to work with:

- image: `torchvision`
- audio: `torchaudio`
- text: `torchtext`
- ...

# Tensors

# Tensors

These are all tensors.



Including those with a rank of 0, 1 and 2, which have common names:  
**scalar, vector, matrix.**

Figure taken from [\[Stevens & Antiga 20\]](#).

## Tensors (cont'd)

You will be manipulating multi-dimensional arrays all the time:

- You have an image.  
It has height, width, and color channels (each pixel represented by red-green-blue): it's a 3d-array.  
You have multiple images: you get a 4d-array.

## Tensors (cont'd)

You will be manipulating multi-dimensional arrays all the time:

- You have an image.  
It has height, width, and color channels (each pixel represented by red-green-blue): it's a 3d-array.  
You have multiple images: you get a 4d-array.
- You have a written sentence.  
You replace each word in the sentence by its ID, you get a 1d-array.  
You have multiple sentences; you get a 2d-array.  
(some subtlety w/ handling various sentence lengths)

## Tensors (cont'd)

You will be manipulating multi-dimensional arrays all the time:

- You have an image.  
It has height, width, and color channels (each pixel represented by red-green-blue): it's a 3d-array.  
You have multiple images: you get a 4d-array.
- You have a written sentence.  
You replace each word in the sentence by its ID, you get a 1d-array.  
You have multiple sentences; you get a 2d-array.  
(some subtlety w/ handling various sentence lengths)
- These are all **tensors**.

Corresponding class in PyTorch is `torch.Tensor`.

Similar to NumPy's `numpy.ndarray`.

# Tensors, basic concepts

`torch.tensor` function creates `torch.Tensor` object:

```
1 >>> import torch
2 >>> x = torch.tensor([[1, 2], [3, 4]])
3 >>> x
4 tensor([[1, 2],
5         [3, 4]])
```

# Tensors, basic concepts

`torch.tensor` function creates `torch.Tensor` object:

```
1 >>> import torch
2 >>> x = torch.tensor([[1, 2], [3, 4]])
3 >>> x
4 tensor([[1, 2],
5         [3, 4]])
```

A fundamental attribute is the **shape** of tensor:

```
1 >>> x.shape # or 'x.size()'
2 torch.Size([2, 2])
```

# Tensors, basic concepts

`torch.tensor` function creates `torch.Tensor` object:

```
1 >>> import torch
2 >>> x = torch.tensor([[1, 2], [3, 4]])
3 >>> x
4 tensor([[1, 2],
5         [3, 4]])
```

A fundamental attribute is the **shape** of tensor:

```
1 >>> x.shape # or 'x.size()'
2 torch.Size([2, 2])
```

Also: every `torch.Tensor` has `dtype` (data type) attribute.

```
1 >>> x.dtype
2 torch.int64
3 >>> x = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
4 >>> x
5 tensor([[1., 2.],
6         [3., 4.]])
7 >>> x.dtype
8 torch.float32
```

## Tensors (cont'd)

You can create some classic/random tensors using e.g.

- eye (diagonal tensor), zeros (all-zero tensor), ones (all-one tensor), arange (1D tensor with incrementally increasing integers), rand (random tensor),... by specifying the shape/size you want.

```
1 >>> x = torch.ones([2, 3])
2 >>> x
3 tensor([[1., 1., 1.],
4         [1., 1., 1.]])
```

```
1 >>> x = torch.rand([2, 3])
2 >>> x
3 tensor([[0.9332, 0.0222, 0.0364],
4         [0.7155, 0.8362, 0.9795]])
```

# Tensors, conversion from/to NumPy ndarrays.

Simply: `torch.from_numpy()` and `x.numpy()`

```
1 >>> import numpy
2 >>> import torch
3 >>> a = numpy.array([1, 2, 3])
4 >>> a
5 array([1, 2, 3])
6 >>> b = torch.from_numpy(a)
7 >>> b
8 tensor([1, 2, 3])
9 >>> b.numpy()
10 array([1, 2, 3])
```

# Tensors, manipulating dimensionality

## Checking the shape of tensor:

- `x.size()` return tuple-like object of shape.
- `x.shape` same as above (alias).
- `x.ndim` return number of dimensions

# Tensors, manipulating dimensionality

## Checking the shape of tensor:

- `x.size()` return tuple-like object of shape.
- `x.shape` same as above (alias).
- `x.ndim` return number of dimensions

## Modifying the shape of tensor:

- `x.view(a,b,...)` return tensor; **reshape** of x to size (a,b,...) without copying.
- `x.reshape(a,b,...)` return tensor; **reshape** of x to size (a,b,...). (when you can not use `view`; you will know as PyTorch will complain).
- `x.permute(*dims)` permute dimensions.
- `x.squeeze(dim)` & `x.unsqueeze(dim)` return tensor with removed/added axis.

# Tensors, manipulating dimensionality

## Checking the shape of tensor:

- `x.size()` return tuple-like object of shape.
- `x.shape` same as above (alias).
- `x.ndim` return number of dimensions

## Modifying the shape of tensor:

- `x.view(a,b,...)` return tensor; **reshape** of x to size (a,b,...) without copying.
- `x.reshape(a,b,...)` return tensor; **reshape** of x to size (a,b,...). (when you can not use `view`; you will know as PyTorch will complain).
- `x.permute(*dims)` permute dimensions.
- `x.squeeze(dim)` & `x.unsqueeze(dim)` return tensor with removed/added axis.

## Combining multiple tensors:

- `torch.cat` , `torch.stack` , to concatenate/ stack multiple tensors... etc!

# Tensors, manipulating dimensionality

## Checking the shape of tensor:

- `x.size()` return tuple-like object of shape.
- `x.shape` same as above (alias).
- `x.ndim` return number of dimensions

## Modifying the shape of tensor:

- `x.view(a,b,...)` return tensor; **reshape** of x to size (a,b,...) without copying.
- `x.reshape(a,b,...)` return tensor; **reshape** of x to size (a,b,...). (when you can not use `view`; you will know as PyTorch will complain).
- `x.permute(*dims)` permute dimensions.
- `x.squeeze(dim)` & `x.unsqueeze(dim)` return tensor with removed/added axis.

## Combining multiple tensors:

- `torch.cat` , `torch.stack` , to concatenate/ stack multiple tensors... etc!

## You will try them all in Exercise 2.

Deep Learning Lab 2022

# Tensors, indexing / slicing

You can access a sub-tensor via indices (similar to NumPy).

```
1 >>> y = torch.arange(15)
2 >>> y
3 tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
4 >>> y.size()
5 torch.Size([15])
6 >>> t = y.view(3,5)
7 >>> t
8 tensor([[ 0,  1,  2,  3,  4],
9         [ 5,  6,  7,  8,  9],
10        [10, 11, 12, 13, 14]])
11 >>> t.size()
12 torch.Size([3, 5])
13 >>> t[1, 3]
14 tensor(8)
15 >>> t[2, 1:3]
16 tensor([11, 12])
17 >>> t[0:2, :]
18 tensor([[ 0,  1,  2,  3,  4],
19         [ 5,  6,  7,  8,  9]])
```

# Tensors, example operations

Different types of **operations**:

- "standard" math operations between two tensors.
- operations which do something on a single tensor; e.g. along some axis/dimension.

```
1  >>> x = torch.randint(low=0, high=10, size=[2, 3])
2  >>> y = torch.randint(low=0, high=10, size=[2, 3])
3  >>> x
4  tensor([[5, 1, 8],
5      [4, 5, 9]])
6  >>> y
7  tensor([[1, 6, 9],
8      [7, 4, 9]])
9  >>> x + y
10 > tensor([[ 6,  7, 17],
11     [11,  9, 18]])
12 >>> x * y
13 > tensor([[ 5,  6, 72],
14     [28, 20, 81]])
```

# Tensors, more example operations

Element-wise function:

```
1 >>> import torch.nn.functional as F
2 >>> z = torch.randint(low=-10, high=10, size=[2, 3])
3 >>> z
4 tensor([[ 0, -7, -4],
5         [-9,  7,  2]])
6 >>> F.relu(z)
7 tensor([[0, 0, 0],
8         [0, 7, 2]])
```

Operation along some axis/dimension (here sum):

```
1 >>> torch.sum(z)
2 tensor(-11)
3 >>> torch.sum(z, axis=0)
4 tensor([-9,  0, -2])
5 >>> torch.sum(z, axis=1)
6 tensor([-11,   0])
```

# Tensors, broadcasting

What happens when sizes do not match?

# Tensors, broadcasting

What happens when sizes do not match?

```
1 >>> a = torch.ones(2,1)
2 >>> a
3 tensor([[1.],
4         [1.]])
5 >>> b = torch.ones(1,2)
6 >>> b
7 tensor([[1., 1.]])
8 >>> a + b
9 tensor([[2., 2.],
10        [2., 2.]])
```

It “guesses” the operation you intend to do: **broadcasting**.

# Tensors, broadcasting

What happens when sizes do not match?

```
1 >>> a = torch.ones(2, 1)
2 >>> a
3 tensor([[1.],
4         [1.]])
5 >>> b = torch.ones(1, 2)
6 >>> b
7 tensor([[1., 1.]])
8 >>> a + b
9 tensor([[2., 2.],
10        [2., 2.]])
```

It “guesses” the operation you intend to do: **broadcasting**.

Make sure that it’s what you really want! Maybe you wanted:

```
1 >>> a + b.t() # b.t() is transpose of b.
2 tensor([[2.],
3         [2.]])
```

<https://pytorch.org/docs/stable/notes/broadcasting.html> to read more about it...

# Tensors on GPU?

General purpose **graphical processing unit**, GPGPU or simply GPU.

- Hardware.
- Accelerate many operations of linear algebra (such as matrix multiplication).
  - Core computations in neural networks.
  - In the past: low level implementation in CUDA had to be done.
  - Now: straightforward using deep learning library (such as PyTorch). Users only need to write code in Python.



# Tensors on GPU? (cont'd)

Again: GPU is a hardware!

- To compute using GPU, you need to copy tensors to GPU memory.
- Every `torch.Tensor` has the `device` attribute (basically CPU or GPU, tells the device on which the tensor is stored.).
- Straightforward to transfer between devices in PyTorch `x.to(device)` :

```

1 >>> device = torch.device(
2 ...     "cuda:0" if torch.cuda.is_available() else "cpu")
3 >>> a = torch.rand([2, 3])
4 >>> a.device
5 device('cpu')
6 >>> b = a.to(device) # copy to GPU.
7 >>> b.device
8 device('cuda', index=0)

```

- You can also directly create a tensor on a specific device  
`a = torch.rand([2, 3], device='cuda')`

NB: `to` is a method introduced after version  $\geq 0.4$ .  
 (Used to be `x.cuda()` and `x.cpu()` before)

## A few words on GPUs

- Different GPUs have different specs. Better GPUs released every year.
- For example, on the USI's ICS cluster:
  - NVIDIA GeForce GTX **1080 8 GB**
  - NVIDIA GeForce GTX **1080 Ti 11 GB**
  - NVIDIA GeForce GTX **2080 Ti 11 GB** (faster than 1080).
- Be aware of the GPU memory of the machine.  
Classic error you will get:  

```
RuntimeError: CUDA out of memory. Tried to allocate ...
```
- A typical solution is to reduce the batch size...
- or run on machines which has more GPU memory.

# Automatic differentiation

Reminder: for gradient descent  $\theta(n+1) = \theta(n) - \alpha * \nabla_{\theta}\mathcal{L}(\theta(n))$   
you need to compute the **gradient** of the loss  $\nabla_{\theta}\mathcal{L}(\theta(n))$  w.r.t. the model parameters. Need to do backpropagation!

In PyTorch, this computation is **automatic**:

- Write computations of a scalar `loss` as a function of some tensors.
- `loss.backward()` computes *all* gradients  
i.e. **no need to write these computations by yourself.**
- *all?* Every `torch.Tensor` has `requires_grad` attribute.

If interested in learning how this works; see e.g.

<https://www.youtube.com/watch?v=MswxJw-8PvE>

# Automatic differentiation (cont'd)

Illustration:

```
1 >>> import torch
2 >>> x = torch.tensor(2, requires_grad=True, dtype=torch.float32)
3 >>> x
4 tensor(2., requires_grad=True)
5 >>> y = torch.tensor(3, requires_grad=True, dtype=torch.float32)
6 >>> z = x * x + y # Forward computation.
7 >>> z.backward() # "dz/dx = 2 x"
8 >>> x.grad
9 tensor(4.)
```

# Automatic differentiation (cont'd)

Note: gradients are accumulated.

Continue from the previous slide:

```
1 >>> a = x * x + y # define one more tensor.  
2 >>> a.backward() # add "da/dx" to "x.grad"  
3 >>> x.grad # contains "dz/dx + da/dx"  
4 tensor(8.)
```

If this is not wanted, you need to reset it:

```
1 >>> x.grad.data.zero_() # reset gradient for x.  
2 tensor(0.)  
3 >>> a = x * x + y  
4 >>> a.backward()  
5 >>> x.grad  
6 tensor(4.)
```

**In practice, you will not even need to do this for each parameter, as you will be using some optimizer.** Examples later.

# Detach a tensor from gradient computation

You can use `detach()` to extract the value of a tensor such that its usage will not influence the gradient computation.

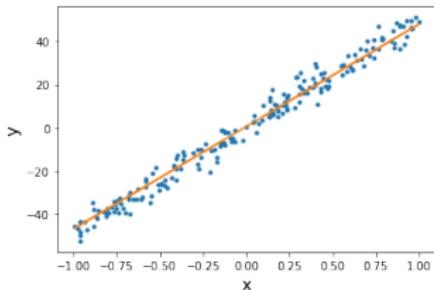
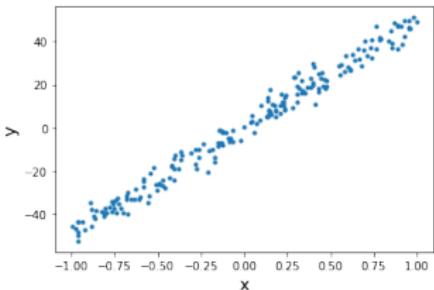
```
1 >>> x = torch.tensor([2., 3.], requires_grad=True)
2 tensor([2., 3.], requires_grad=True)
3 >>> y = x.detach()
4 tensor([2., 3.])
```

# Example problem solving

## Regression task:

- Let  $D$  be a positive integer. We have  $N$  i.i.d. data points  $\{(x_1, y_1), \dots, (x_N, y_N)\}$  where  $x_i \in \mathbb{R}^D$ ,  $y_i \in \mathbb{R}$  for all  $1 \leq i \leq N$ .
- We want to find a function  $f_\theta$  parameterized by  $\theta$  (a set of real numbers), which predicts  $y$  from unseen  $x$ .
- **Linear regression:** use  $f_\theta(x) = w^\top x + b$  where  $w \in \mathbb{R}^D$  and  $b \in \mathbb{R}$  are **model parameters**. So  $\theta = \{w, b\}$ .

**Illustration for  $D = 1$ .**



$D = 1$  in all what follows.

# Linear regression (with $D = 1$ )

- We have  $N$  i.i.d. data points  $\{(x_1, y_1), \dots, (x_N, y_N)\}$  where  $x_i \in \mathbb{R}$   $y_i \in \mathbb{R}$  for all  $1 \leq i \leq N$ .
- Our linear model  $f_{w,b}(x) = w * x + b$  has two parameters:  $w \in \mathbb{R}$  and  $b \in \mathbb{R}$ .
- We estimate these **parameters** by minimizing **mean squared error (MSE)** between model predictions and the actual data through **gradient descent**.

# Linear regression (with $D = 1$ )

- We have  $N$  i.i.d. data points  $\{(x_1, y_1), \dots, (x_N, y_N)\}$  where  $x_i \in \mathbb{R}$   $y_i \in \mathbb{R}$  for all  $1 \leq i \leq N$ .
- Our linear model  $f_{w,b}(x) = w * x + b$  has two parameters:  $w \in \mathbb{R}$  and  $b \in \mathbb{R}$ .
- We estimate these **parameters** by minimizing **mean squared error (MSE)** between model predictions and the actual data through **gradient descent**.

## Algorithm:

- Randomly initialize the model parameters:  $w_0 \in \mathbb{R}$  and  $b_0 \in \mathbb{R}$ .
- Choose **hyper-parameters**: learning rate  $\alpha$  & number of training steps  $T$ .
- Repeat the following training steps  $T$  times. At each step  $t$  (from 0 to  $T - 1$ ):
  - Compute the MSE loss using the current value of  $w_t$  and  $b_t$ :

$$\mathcal{L}(w_t, b_t) = \frac{1}{N} \sum_{i=1}^N \|f_{w_t, b_t}(x_i) - y_i\|^2$$

- Compute the corresponding gradients:  $\frac{\partial \mathcal{L}}{\partial w}(w_t, b_t)$  and  $\frac{\partial \mathcal{L}}{\partial b}(w_t, b_t)$
- Update parameters:  $w_{t+1} = w_t - \alpha \frac{\partial \mathcal{L}}{\partial w}(w_t, b_t)$  and  $b_{t+1} = b_t - \alpha \frac{\partial \mathcal{L}}{\partial b}(w_t, b_t)$

Code? PyTorch?

# Linear regression

## How to generate data points?

This is a “toy task.” We synthetically generate  $N$  data points:

- For that, we start by setting the "true"  $w^*$  and  $b^*$  (that we choose).
- We randomly sample  $x_i$ . Using that we compute  $\hat{y}_i = w * x_i + b$ .
- Add a Gaussian noise:  $y_i = \hat{y}_i + \epsilon_i$  with  $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$  where  $\sigma$  is the standard deviation (that we choose).
- We end up with noisy data points:  $(x_i, y_i)$

```
1 import numpy as np
2
3 def create_dataset(sample_size=10, sigma=0.1, w_star=1, b_star=1,
4                     x_range=(-1, 1), seed=0):
5     """Create data points for linear regression."""
6     random_state = np.random.RandomState(seed)
7     x_min, x_max = x_range
8     x = random_state.uniform(x_min, x_max, (sample_size))
9     y = x * w_star + b_star
10    y += random_state.normal(0.0, sigma, (sample_size))
11
12    return x, y
```

# Linear regression (cont'd)

## How to generate data points?

- Generate the training data:

```
1 num_samples = 200
2 sigma = 4
3 w_star = 50
4 X, y = create_dataset(
5     sample_size=num_samples, sigma=sigma, w_star=w_star, seed=0)
```

- From the same distribution, also sample the **validation** data points using a different **seed**:

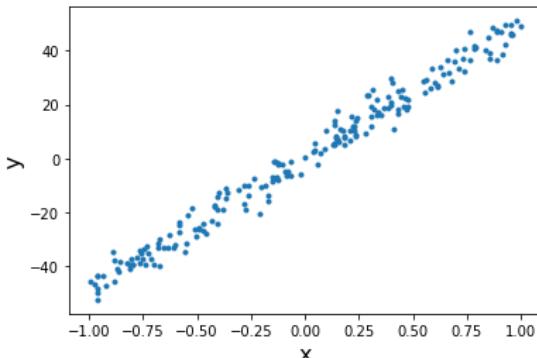
```
1 val_num_samples = 200
2 X_val, y_val = create_dataset(
3     sample_size=num_samples, sigma=sigma, w_star=w_star, seed=42)
```

# Linear regression (cont'd)

## How to generate data points?

- Visualize the data:

```
1 import matplotlib.pyplot as plt
2
3 fig, ax = plt.subplots()
4 ax.set_xlabel("x", fontsize=16)
5 ax.set_ylabel("y", fontsize=16)
6
7 ax.plot(X, y, ".")
```



Check the documentation!

# Linear regression, basic components

All necessary components are already implemented in PyTorch.

- We need a linear model  $f: f_{\theta}(x) = w * x + b$   
where  $w \in \mathbb{R}$  and  $b \in \mathbb{R}$  are the model parameters. Build-in model in PyTorch is `torch.nn.Linear`.
- We need the mean squared error: `torch.nn.MSELoss`.
- We update the model parameters using gradient descent. The corresponding optimizer class is `torch.optim.SGD`

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4
5 DEVICE = torch.device("cuda:0" if torch.cuda.is_available()
6                         else "cpu")
7
8 model = nn.Linear(1, 1) # input dimension 1, and output dimension 1.
9 model = model.to(DEVICE)
10 loss_fn = nn.MSELoss()
11 learning_rate = 0.1
12 optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

# Linear regression (cont'd)

Illustration of components (try on your own!):

■ `torch.nn.Linear`.

```
1 >>> import torch
2 >>> model = torch.nn.Linear(1, 1)
3 >>> model.weight
4 Parameter containing:
5 tensor([-0.6464]), requires_grad=True)
6 >>> model.bias
7 Parameter containing:
8 tensor([-0.7324], requires_grad=True)
9 >>> x = torch.tensor([3.])
10 >>> model(x)
11 # -0.6464 * 3. - 0.7324
12 tensor([-2.6717], grad_fn=<AddBackward0>)
```

- Recommendation: read documentation of `torch.nn.Linear`.
- e.g., how are the parameters initialized by default?

# Linear regression (cont'd)

## ■ torch.nn.MSELoss .

```
1 >>> import torch
2 >>> loss = torch.nn.MSELoss()
3 >>> a = torch.tensor([2.])
4 >>> b = torch.tensor([2.2])
5 >>> loss(a, b)
6 tensor(0.0400)
```

## ■ torch.optim.SGD

```
1 >>> import torch
2 >>> x = torch.tensor(2., requires_grad=True)
3 >>> z = x * x
4 >>> opt = torch.optim.SGD([x], lr=0.1)
5 >>> z.backward()
6 >>> opt.step()
7 >>> x
8 tensor(1.6000, requires_grad=True)
```

Again: **try on your own!** read documentation.

# Linear regression (cont'd)

- Back to our regression problem.
- The rest is to run the training loop.
- But before that, data shape/dtype/device must be prepared as expected by the function which will take them as input.

```
1 X = X.reshape(num_samples, 1) # shape expected by nn.Linear
2 X = torch.from_numpy(X) # convert to torch.tensor
3 X = X.float() # convert to float32 (from numpy double).
4 X = X.to(DEVICE) # copy data to GPU.
5
6 # Same for other variables:
7 y = torch.from_numpy(y.reshape((num_samples, 1))).float().to(DEVICE)
8 X_val = torch.from_numpy(
9     X_val.reshape((val_num_samples, 1))).float().to(DEVICE)
10 y_val = torch.from_numpy(
11     y_val.reshape((val_num_samples, 1))).float().to(DEVICE)
```

# Linear regression (cont'd)

Run the training loop.

```
1 num_steps = 50 # We do 50 steps of gradient updates.  
2  
3 for step in range(num_steps):  
4     model.train() # systematic: put model in 'training' mode.  
5     optimizer.zero_grad() # systematic: start step w/ zero gradient.  
6  
7     y_ = model(X) # do prediction using the current model.  
8     loss = loss_fn(y_, y) # compute error.  
9     print(f"Step {step}: train loss: {loss}") # print train loss  
10  
11    loss.backward() # compute gradients.  
12    optimizer.step() # update parameters  
13  
14    # Eval on validation set  
15    model.eval() # systematic: put model in 'eval' mode.  
16    # everything below does not contribute to gradient computation  
17    with torch.no_grad():  
18        y_ = model(X_val)  
19        val_loss = loss_fn(y_, y_val)  
20        print(f"Step {step}: val loss: {val_loss}")
```

Or: `model.zero_grad()` instead of `optimizer.zero_grad()`.

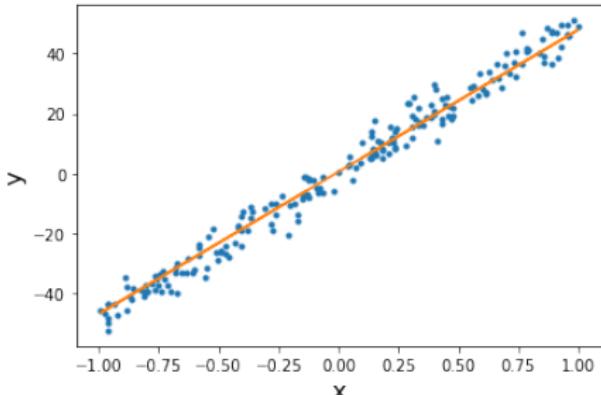
# Linear regression (cont'd)

Plot the resulting model:

```

1 # Get the prediction from the final model.
2 model.eval()
3 with torch.no_grad():
4     y_ = model(X)
5 fig, ax = plt.subplots()
6 ax.plot(X.cpu().numpy(), y_.cpu().numpy(), ".")
7 ax.plot(X.cpu().numpy(), y_.cpu().numpy(), "-")
8
9 ax.set_xlabel("x", fontsize=16)
10 ax.set_ylabel("y", fontsize=16)

```



# Linear regression (cont'd)

We can also take a look at the final model parameters:

```
1 >> model.weight
2 Parameter containing:
3 tensor([[47.3805]], device='cuda:0', requires_grad=True)
4 >> model.bias
5 Parameter containing:
6 tensor([0.5539], device='cuda:0', requires_grad=True)
```

- This was a simple example problem to illustrate some PyTorch functionalities.
- A similar problem is to be solved in the [Assignment 1](#).
- In the next lectures, we look into more details of components for building systems.

## A few words on Exercise 2 & 3

### Now: Exercise 2

- Objective: get familiar with PyTorch basics.

### Next week: Assignment 1 (Exercise 3)

- First assignment
- Contributes to 15% of the final grade.
- Task: polynomial regression
- You will be using many basic tools that you have learned:
  - numpy random number generator
  - plot data points
  - ...

# Outline

1. Workflow Overview
- 2. Introduction to Tools**
  1. Basic Tools
  2. Implementation of Systems
3. Fundamental building blocks
4. Building models
5. Practical tricks and methods for training
6. Final words and Outlook

## What do we need? Reminder:

- Dataset
- Model
- Loss
- Optimizer
- Multiple optimization steps (training loop).

# Structure Preview: Sketch

```
1 device = torch.device('cuda') # 'cuda' for GPU.  
2 dataset = MyDataset(...)  
3 dataloader = DataLoader(dataset, ...)  
4  
5 model = MyModel(...)  
6 model = model.to(device) # put the model params on device.  
7 optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)  
8 loss_fn = Loss(...)  
9  
10 for i in range(10): # Training loop for 10 epochs.  
11     for sample in dataloader: # get batch.  
12         data = sample['data'].to(device)  
13         target = sample['target'].to(device)  
14  
15         prediction = model(data)  
16         loss = loss_fn(prediction, target)  
17  
18         optimizer.zero_grad() # reset the gradients  
19         loss.backward()  
20         optimizer.step() # update params.  
21     # validation.  
22         # compute validation loss ...  
23         # save the intermediate model, ...
```

# Implementing datasets / dataloader

For SGD, we need to generate data batches (i.e. randomly sampled training examples).

## Two useful classes in `torch.utils.data`

- `torch.utils.data.Dataset` : class for representing the dataset.
- `torch.utils.data.DataLoader` : class for generating batches of data (does e.g. shuffling).

Official tutorial:

[https://pytorch.org/tutorials/beginner/data\\_loading\\_tutorial.html](https://pytorch.org/tutorials/beginner/data_loading_tutorial.html)

# Dataset

## Three main methods:

- `__init__(self)`
- `__getitem__(self, index)`
- `__len__(self)`

```
1 from torch.utils.data import Dataset, DataLoader
2
3 class RandomDataset(Dataset):
4
5     def __init__(self, data_dim, num_data_points):
6         self.len = num_data_points
7         self.data = torch.randn(num_data_points, data_dim)
8
9     def __getitem__(self, index):
10        return self.data[index]
11
12    def __len__(self):
13        return self.len
```

# Dataloader

Generate batches.

Continue from the previous slides:

```
1 input_dim = 10
2 data_size = 1000
3 rand_dataset = RandomDataset(input_dim, data_size)
4
5 batch_size = 32
6 data_loader =
7     DataLoader(dataset=rand_dataset,
8                 batch_size=batch_size, shuffle=True)
9
10 for i, data in enumerate(data_loader, 0):
11     # each data contains 32 random samples of the data
12     # data can then be fed to the model.
13     ...
```

One can also specify a more specific way of sampling batches e.g. via `sampler` argument of `DataLoader`. Alternatively: you could also implement your own data loader.

# Implementing models using PyTorch

PyTorch **models** are regular Python class which inherits from `torch.nn.Module`

## Two main methods

- `__init__(self)` : define model components.
- `forward(self, x)` : forward computation, given input  $x$ .

# PyTorch Models, example

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 class MyModel(nn.Module):
5
6     def __init__(self, input_size, output_size):
7         super(MyModel, self).__init__()
8         self.fc = nn.Linear(input_size, output_size)
9
10    def forward(self, input):
11        output = self.fc(input)
12        output = F.relu(output)
13        return output
```

Note: model itself can contain `nn.Module` object, here `nn.Linear` a linear transformation layer.

These basic building blocks are already implemented; ready to be used.

# Checking model information

# Checking model information

Continue from last slide.

`print` gives a decent overview:

```
1 >>> model = MyModel(5, 2)
2 >>> print(model)
3 MyModel(
4     (fc): Linear(in_features=2, out_features=5, bias=True)
5 )
```

# Checking model information

Continue from last slide.

`print` gives a decent overview:

```
1 >>> model = MyModel(5, 2)
2 >>> print(model)
3 MyModel(
4     (fc): Linear(in_features=2, out_features=5, bias=True)
5 )
```

Inspecting model parameters via `state_dict()`:

```
1 >>> for key, value in model.state_dict().items():
2 ...     print(key)
3 ...     print(value)
4 fc.weight
5 tensor([[ 0.1786,   0.4163, -0.0949,   0.0152,   0.4388],
6         [-0.2598,   0.0871,   0.3840, -0.3750,   0.4289]])
7 fc.bias
8 tensor([0.0083,  0.1554])
```

# Train vs. Evaluation Modes

Some model components have different behaviors whether it is in training or evaluation mode (e.g. dropout, batch normalization).

# Train vs. Evaluation Modes

Some model components have different behaviors whether it is in training or evaluation mode (e.g. dropout, batch normalization).

- Run `model.train()` before training.
- Run `model.eval()` before evaluation.

# Train vs. Evaluation Modes

Some model components have different behaviors whether it is in training or evaluation mode (e.g. dropout, batch normalization).

- Run `model.train()` before training.
- Run `model.eval()` before evaluation.

Also, to avoid modifying your model during evaluation, you should everything under:

```
1 with torch.no_grad():
2     # evaluation
3     ...
```

to prevent gradient computation (also saves memory).

Different types of losses are available:

- `torch.nn.MSELoss`
- `torch.nn.CrossEntropyLoss` (note: softmax is included in the loss computation)
- etc...

Illustration:

```
1 >>> loss_fn = nn.MSELoss()  
2 >>> predictions = torch.randn(3, 5, requires_grad=True)  
3 >>> targets = torch.randn(3, 5)  
4 >>> loss = loss_fn(predictions, targets)  
5 >>> loss.backward()
```

# Optimizer

Very much straightforward:

- Choose optimizer of your choice.  
Most popular ones: `torch.optim.Adam` and `torch.optim.SGD`.
- Do `optimizer.zero_grad()` before backward computation.  
Remember the gradients are accumulated.
- Do `optimizer.step()`. It updates all params.

```
1  optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)
2
3  for sample in dataloader:
4      data = sample['input'].to(device)
5      target = sample['target'].to(device)
6      prediction = model(data)
7      loss = loss_fn(prediction, target)
8      optimizer.zero_grad()    # reset gradients.
9      loss.backward()
10     optimizer.step()    # one step of update according to optimizer.
```

# Saving & Loading models

## ■ torch.save

```
1 state = {'model_state' : model.state_dict(),  
2           'optimizer': optimizer.state_dict())  
3 torch.save(state, 'state.pt')
```

## ■ torch.load

```
1 model = MyModel()  
2 optimizer = optim.SGD(model.parameters(), lr=0.01)  
3 checkpoint = torch.load('state.pt')  
4 model.load_state_dict(checkpoint['model_state'])  
5 optimizer.load_state_dict(checkpoint['optimizer_state'])
```

Note: some optimizers also have some states which must be stored, in order to resume training from where it had been stopped.

# Monitoring training

Finally, all pieces are essentially there!

# Monitoring training

Finally, all pieces are essentially there!

One last thing: **monitoring training process.**

# Monitoring training

Finally, all pieces are essentially there!

One last thing: **monitoring training process.**

- Training can take a long time. Depending on the task, it can take from a few days to a few months!  
(fortunately not in our exercises.)

# Monitoring training

Finally, all pieces are essentially there!

One last thing: **monitoring training process.**

- Training can take a long time. Depending on the task, it can take from a few days to a few months!  
(fortunately not in our exercises.)
- You would want to check the state of its progress by *monitoring* whether the training and validation losses are **going down**.

# Monitoring training

Finally, all pieces are essentially there!

One last thing: **monitoring training process.**

- Training can take a long time. Depending on the task, it can take from a few days to a few months!  
(fortunately not in our exercises.)
- You would want to check the state of its progress by *monitoring* whether the training and validation losses are **going down**.
- Also, you can detect very bad choices of (training) hyper-parameters, by checking the values of the training loss **just after a few updates!**

→ **You will experience this in the exercises.**

# Monitoring training

Finally, all pieces are essentially there!

One last thing: **monitoring training process**.

- Training can take a long time. Depending on the task, it can take from a few days to a few months!  
(fortunately not in our exercises.)
- You would want to check the state of its progress by *monitoring* whether the training and validation losses are **going down**.
- Also, you can detect very bad choices of (training) hyper-parameters, by checking the values of the training loss **just after a few updates!**

→ **You will experience this in the exercises.**

Methods for monitoring:

- **Print the loss values** regularly, after each  $n$  updates.
- Use **visualization tools**: tensorboard, *Weights & Biases*, ...

# Tensorboard

- Tool for visualizing and monitoring training.
- Originally part of TensorFlow (Google).

# Tensorboard

- Tool for visualizing and monitoring training.
- Originally part of TensorFlow (Google).

Basic idea:

- Add a few lines in your code to record the quantities of your interest in a log.
- `tensorboard` provides visualization of the log (on a browser).

# Tensorboard

- Tool for visualizing and monitoring training.
- Originally part of TensorFlow (Google).

Basic idea:

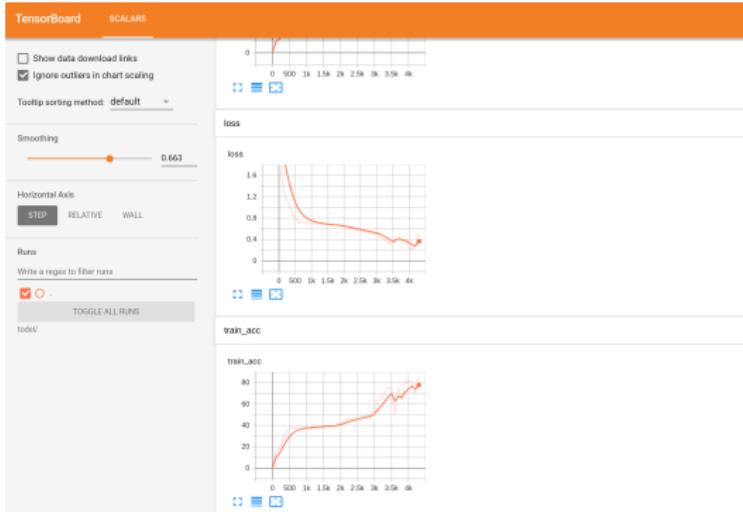
- Add a few lines in your code to record the quantities of your interest in a log.
- tensorboard provides visualization of the log (on a browser).

```
1 from tensorflow import SummaryWriter
2
3 writer = SummaryWriter(logdir='my-experiment/')
4
5 # ... inside the training loop ...
6     # 'train_steps' counts the training steps.
7     # forward the model, compute loss etc
8     writer.add_scalar('loss', loss, train_steps)
9     #
10    writer.add_scalar('train acc', train_acc, train_steps)
11    # ... etc
```

(or `from torch.utils.tensorboard import SummaryWriter`)

# Tensorboard, illustration

- You need to first install tensorboard: `pip install tensorflow`
- Run `tensorboard --logdir my-experiment` on terminal.
- This should give:  
 TensorBoard 2.3.0 at `http://localhost:6006/`
- Paste it on your browser:



# Put them all together! Full example

Let's consider **digit image classification** task using the MNIST dataset [LeCun & Cortes<sup>+</sup> 98] and a two-layer **feed-forward neural network**:

# Put them all together! Full example

Let's consider **digit image classification** task using the MNIST dataset [LeCun & Cortes<sup>+</sup> 98] and a two-layer **feed-forward neural network**:

- Task: **classification** of images; each image is one handwritten digit (0 to 9).
  - Input to the model: image.
  - Output of the model: 10-dim vector, representing probability for each digit (10 classes).
  - Can feed-forward models handle this problem? Yes.

# Put them all together! Full example

Let's consider **digit image classification** task using the MNIST dataset [LeCun & Cortes<sup>+</sup> 98] and a two-layer **feed-forward neural network**:

- Task: **classification** of images; each image is one handwritten digit (0 to 9).
  - Input to the model: image.
  - Output of the model: 10-dim vector, representing probability for each digit (10 classes).
  - Can feed-forward models handle this problem? Yes.

Basic statistics:

- 60 K training images, 10 K test images.
- Dimension of each image:  $\mathbb{R}^{28 \times 28}$ , in black & white (one channel),
  - which can be reshaped to 784-dim vector, to input to the model.

# Import packages... etc

```
1 import torch
2 import torch.nn as nn
3 import torchvision
4 import torchvision.transforms as transforms

1 device = torch.device(
2     "cuda:0" if torch.cuda.is_available() else "cpu")

1 # Hyper-parameters
2 # to specify model:
3 input_size = 784
4 hidden_size = 512
5 num_classes = 10
6
7 # for training:
8 num_epochs = 5
9 batch_size = 128
10 learning_rate = 0.001
11 momentum = 0.9
```

# Create datasets/dataloader

MNIST is standard dataset: you can directly get it from torchvision.

```
1 # Datasets
2 train_set = torchvision.datasets.MNIST(
3     root='./data', train=True, transform=transforms.ToTensor(),
4     download=True)
5
6 test_set = torchvision.datasets.MNIST(
7     root='./data', train=False, transform=transforms.ToTensor())
8
9 # Dataloaders
10 train_loader = torch.utils.data.DataLoader(
11     dataset=train_set, batch_size=batch_size, shuffle=True)
12
13 test_loader = torch.utils.data.DataLoader(
14     dataset=test_set, batch_size=batch_size, shuffle=False)
```

## Define model

- We use a basic **feed-forward** neural network with 2 layers (one hidden vector) with the ReLU activation function.
  - The dimension  $d$  of the hidden vector is the **model hyper-parameter**.
  - The input and output dimensions are specified by the problem.

# Define model

- We use a basic **feed-forward** neural network with 2 layers (one hidden vector) with the ReLU activation function.
  - The dimension  $d$  of the hidden vector is the **model hyper-parameter**.
  - The input and output dimensions are specified by the problem.
- Input image  $x \in \mathbb{R}^{784}$  (reshaped 28x28 image).
- The first layer transforms  $x \in \mathbb{R}^{784}$  to a hidden vector  $h \in \mathbb{R}^d$ :

$$h = \text{ReLU}(W_1x + b_1) = \max(W_1x + b_1, 0)$$

$W_1 \in \mathbb{R}^{d \times 784}$  and  $b_1 \in \mathbb{R}^d$  are **trainable/model parameters**.

# Define model

- We use a basic **feed-forward** neural network with 2 layers (one hidden vector) with the ReLU activation function.
- The dimension  $d$  of the hidden vector is the **model hyper-parameter**.
- The input and output dimensions are specified by the problem.
- Input image  $x \in \mathbb{R}^{784}$  (reshaped 28x28 image).
- The first layer transforms  $x \in \mathbb{R}^{784}$  to a hidden vector  $h \in \mathbb{R}^d$ :

$$h = \text{ReLU}(W_1x + b_1) = \max(W_1x + b_1, 0)$$

$W_1 \in \mathbb{R}^{d \times 784}$  and  $b_1 \in \mathbb{R}^d$  are **trainable/model parameters**.

- The output layer transforms  $h \in \mathbb{R}^d$  to a vector  $y \in \mathbb{R}^{10}$  (size = number of classes = 10 digit):

$$y = W_2h + b_2$$

$W_2 \in \mathbb{R}^{10 \times d}$  and  $b_2 \in \mathbb{R}^{10}$  are **trainable/model parameters**.

# Define model

- We use a basic **feed-forward** neural network with 2 layers (one hidden vector) with the ReLU activation function.
- The dimension  $d$  of the hidden vector is the **model hyper-parameter**.
- The input and output dimensions are specified by the problem.
- Input image  $x \in \mathbb{R}^{784}$  (reshaped 28x28 image).
- The first layer transforms  $x \in \mathbb{R}^{784}$  to a hidden vector  $h \in \mathbb{R}^d$ :

$$h = \text{ReLU}(W_1x + b_1) = \max(W_1x + b_1, 0)$$

$W_1 \in \mathbb{R}^{d \times 784}$  and  $b_1 \in \mathbb{R}^d$  are **trainable/model parameters**.

- The output layer transforms  $h \in \mathbb{R}^d$  to a vector  $y \in \mathbb{R}^{10}$  (size = number of classes = 10 digit):

$$y = W_2h + b_2$$

$W_2 \in \mathbb{R}^{10 \times d}$  and  $b_2 \in \mathbb{R}^{10}$  are **trainable/model parameters**.

- Softmax function transforms the logit  $y$  to a probability distribution: for  $i \in \{0, \dots, 9\}$

$$p(i|x) = \text{softmax}(y)_i$$

# Define model, implementation

```
1 class FFModel(nn.Module):
2
3     def __init__(self, input_size, hidden_size, num_classes):
4         super(FFModel, self).__init__()
5         self.fc1 = nn.Linear(input_size, hidden_size)
6         self.fc2 = nn.Linear(hidden_size, num_classes)
7
8     def forward(self, x):
9         # Assume linealized input: here `images.view(-1, 28*28)` .
10        out = self.fc1(x)
11        out = F.relu(out)
12        out = self.fc2(out)
13        # second layer does not need activation function;
14        # softmax is computed by cross entropy loss.
15        return out
```

# Create model, loss, optimizer

```
1 # Create model
2 model = FFModel(input_size, hidden_size, num_classes)
3 model = model.to(device) # put all model params on GPU.
4
5 # Create loss and optimizer
6 loss_fn = nn.CrossEntropyLoss()
7
8 optimizer = optim.SGD(
9     model.parameters(), lr=learning_rate, momentum=momentum)
```

# Training

```
1 # Training
2 for epoch in range(num_epochs):
3     model.train() # Set model in train mode.
4     for i, (images, labels) in enumerate(train_loader):
5         # shape of images is (B, 1, 28, 28).
6         images = images.view(-1, 28*28) # reshape to (B, 784).
7         images = images.to(device) # copy data to GPU.
8         labels = labels.to(device) # shape (B).
9
10        outputs = model(images) # shape (B, 10).
11        loss = loss_fn(outputs, labels)
12
13        optimizer.zero_grad() # reset gradients.
14        loss.backward() # compute gradients.
15        optimizer.step() # update parameters.
16
17        # here, print the current value of loss...
18        # saving model checkpoint... to be done in Exercise!
```

# Evaluation

```
1 # Evaluation
2 with torch.no_grad():
3     correct = 0
4     total = 0
5     model.eval() # Set model in eval mode. Don't forget!
6     for images, labels in test_loader:
7         images = images.view(-1, 28*28)
8         images = images.to(device)
9         labels = labels.to(device) # shape (B)
10        outputs = model(images) # shape (B, num_classes)
11        # 'outputs' are logits (unnormalized log prob).
12        # Model prediction is the class which has the highest
13        # probability according to the model,
14        # i.e. the class which has the highest logit value:
15        _, predicted = outputs.max(dim=1)
16        # predicted.shape: (B)
17        total += labels.size(0)
18        correct += (predicted == labels).sum().item()
19        test_acc = 100 * correct / total
20
21 print(f'Test accuracy is: {test_acc} %')
```

# Not covered in this example

- Preparation of **validation** dataset!  
Crucial for tuning **hyper-parameter**.
- Hyper-parameter tuning.
- Monitoring of intermediate model performance during training & saving model parameters.
- ... we will see more refinements to in the chapter after next one!

You will do these in **Exercise 4!**

# Excursions: C++ API

**Beyond the scope of this lab.**

When it is needed to go beyond Python?

- To write custom operations in C++ to optimize some special operations/models in terms of speed or memory consumption.

Official example: *long long-term memory*

[https://pytorch.org/tutorials/advanced/cpp\\_extension.html](https://pytorch.org/tutorials/advanced/cpp_extension.html)

- To deploy models in C++/production code: you trained your model using code in Python but want to use the trained model in other code written in C++. Example Tesla?

[https://pytorch.org/tutorials/advanced/cpp\\_export.html](https://pytorch.org/tutorials/advanced/cpp_export.html)

## What have we learned?

- Basics of Python+PyTorch.
- Basic implementation of main components of deep learning workflow using PyTorch.
- Many examples.

## Coming up next...

- What kind of neural networks should we consider for different problems?
- How are they implemented in PyTorch?
- ...

# Outline

1. Workflow Overview
2. Introduction to Tools
- 3. Fundamental building blocks**
  1. Feed-forward NNs, Convolutional NNs, Residual NNs
  2. Recurrent NNs and LSTM
  3. Attention, Self-attention, Transformers
4. Building models
5. Practical tricks and methods for training
6. Final words and Outlook

# Outline

1. Workflow Overview
2. Introduction to Tools
- 3. Fundamental building blocks**
  1. Feed-forward NNs, Convolutional NNs, Residual NNs
  2. Recurrent NNs and LSTM
  3. Attention, Self-attention, Transformers
4. Building models
5. Practical tricks and methods for training
6. Final words and Outlook

# Different types of neural networks

- There are **different types of neural networks**, which are designed for different problems/sub-problems.
- For those of you attending the Machine Learning lecture:  
you have seen/will learn these models.
- In this course: overview/reminder with **example PyTorch code**.
  - Opportunity to make sure you understand: input/output shapes, model parameters, ... how models work!
- Outline:
  - (Feed-forward neural networks): Exercise 4
  - Convolutional neural networks
  - Recurrent neural networks (RNNs), and long short-term memory (LSTM)
  - Neural attention, self-attention, and Transformers.

The assignments will cover concrete applications.

# Feed-forward neural networks

We have already seen **feed-forward neural networks** in the previous chapter:

- It's the basic model/layer to map/transform vectors.
- It is also referred to as **multi-layer perceptron (MLP)**.  
Even when it only has one hidden layer.
- The main transformation is also referred to as a **fully connected** layer  
(in contrast e.g. to a convolutional layer).
- Terminology: *feed-forward neural network* is a generic term for all neural networks which are **not recurrent**.

# Convolutional neural networks

Typical shorthands: Convolutional nets, ConvNets, CNN.

Typical convNets consist of a **stack** of:

- **Convolutional layers**
- **Pooling layers** (e.g. max-pooling)
- Fully connected layers

# Convolutional layers, motivation

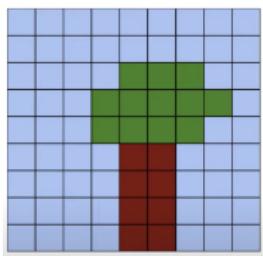
- Remember what we did in the previous chapter?  
For image classification using an MLP, we **reshaped input images to vectors**, and applied a linear transformation to them.
- Does this make sense? You might answer:
  - Yes. We do not care, just plug in any input vectors, connect the output to a loss, and it automatically learns the correlation/regularity. It's deep learning!
  - No. We should exploit properties of images to *facilitate learning* (jargon: we introduce an *inductive bias* to the model architecture)

Which one sounds more reasonable? at this stage both?

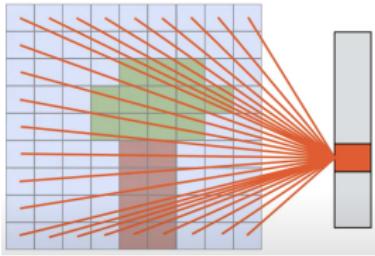
# Convolutional layers, motivation

- Remember what we did in the previous chapter?  
For image classification using an MLP, we **reshaped input images to vectors**, and applied a linear transformation to them.
- Does this make sense? You might answer:
  - Yes. We do not care, just plug in any input vectors, connect the output to a loss, and it automatically learns the correlation/regularity. It's deep learning!
  - No. We should exploit properties of images to *facilitate learning* (jargon: we introduce an *inductive bias* to the model architecture)Which one sounds more reasonable? at this stage both?
- Some (useful) properties of images/for image recognition
  - Locality (pixels which are close, likely belong to the same pattern): local connection.
  - Translation equivariance: we want to recognize patterns in an image independent of their position.

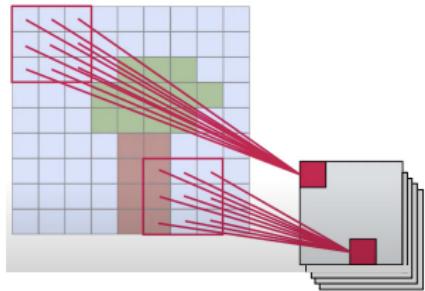
# Convolutional layers, illustration



Input image.



Fully connected layer.



Convolutional layer.

Figures adapted from [Dieleman 20].

Check out more animated illustrations:

- [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)
- <https://cs231n.github.io/convolutional-networks/>

Also: lecture by Sander Dieleman (Deepmind) at UCL, London:

<https://www.youtube.com/watch?v=shVKh0mTOHE&feature=youtu.be>

# Convolutional layers, operation

Let  $C, H, W, C', H', W', d_1, d_2$  denote positive integers.

Description for 2D case. 2D convolutional layer:

- **input:** image-like tensor  $x$  with shape  $(C, H, W)$  (let's omit batch for now)
- **output:** also image-like  $y$  with shape  $(C', H', W')$ , called *feature map*
- **model parameters:** weights and biases!
  - ConvNets have  $C'$  **kernel/filters** (vs. Linear layers' weight matrix).
    - $C'$  = number of output channels
  - Each kernel is a tensor of size  $(C, d_1, d_2)$ . Here let's assume square kernels  $d = d_1 = d_2$ . They are small "image templates".
  - bias  $b \in \mathbb{R}^{C'}$

# Convolutional layers, operation

Let  $C, H, W, C', H', W', d_1, d_2$  denote positive integers.

Description for 2D case. 2D convolutional layer:

- **input:** image-like tensor  $x$  with shape  $(C, H, W)$  (let's omit batch for now)
- **output:** also image-like  $y$  with shape  $(C', H', W')$ , called *feature map*
- **model parameters:** weights and biases!
  - ConvNets have  $C'$  **kernels/filters** (vs. Linear layers' weight matrix).
  - $C'$  = number of output channels
  - Each kernel is a tensor of size  $(C, d_1, d_2)$ . Here let's assume square kernels  $d = d_1 = d_2$ . They are small "image templates".
  - bias  $b \in \mathbb{R}^{C'}$

Given **input**  $x \in \mathbb{R}^{C \times H \times W}$ , for each each output channel  $1 \leq k \leq C'$ , **output**  $y_{k,i,j}$  ( $1 \leq i \leq H'$ , and  $1 \leq j \leq W'$ ) is computed using the corresponding **kernel**  $f^{(k)}$  of size  $d$  (i.e.  $f^{(k)} \in \mathbb{R}^{C \times d \times d}$ ):

$$y_{k,i,j} = \sigma \left( b_k + \sum_{c=1}^C \sum_{i'=i}^{i+d-1} \sum_{j'=j}^{j+d-1} f_{c,i'-i+1,j'-j+1}^{(k)} \times x_{c,i',j'} \right)$$

# Convolutional layers, operation (cont'd)

- Indices make it look complicated, but it is not!  
We are sliding each kernel on the input image.  
See e.g. [https://github.com/vdumoulin/conv\\_arithmetic/blob/master/gif/no\\_padding\\_no\\_strides.gif](https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/no_padding_no_strides.gif)
- At each position, for each kernel, we are simply computing the **dot product** (similarity measure) between the kernel and the input image within the local window.
- Some terminologies:
  - Locally connected (as opposed to fully connected)
  - Weight sharing (across positions).
- Note: computing a similarity between two “vectors” using dot product is a fundamental concept (we will also see this for computing *attention*).

# Convolutional layers, specifications

To fully define the convolutional layer, we have to specify:

- number of input/output channels
- kernel size
- padding: how many "zeros" (or ones?) do we add on the borders to adjust the output "image" size?
- [https://github.com/vdumoulin/conv\\_arithmetic/blob/master/gif/same\\_padding\\_no\\_strides.gif](https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/same_padding_no_strides.gif)
- stride: how many positions do we skip when we move the kernel over the image? also influence output size.

[https://github.com/vdumoulin/conv\\_arithmetic/blob/master/gif/padding\\_strides.gif](https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/padding_strides.gif)

See options in PyTorch `nn.Conv2d`. No need to overthink.

Docs > [torch.nn](#) > Conv2d



## CONV2D

```
CLASS torch.nn.Conv2d(in_channels: int, out_channels: int, kernel_size: Union[T,  
Tuple[T, T]], stride: Union[T, Tuple[T, T]] = 1, padding: Union[T, Tuple[T,  
T]] = 0, dilation: Union[T, Tuple[T, T]] = 1, groups: int = 1, bias: bool =  
True, padding_mode: str = 'zeros')
```

[SOURCE]

## More options in `nn.Conv2d` (excursion)

- groups: number into which the input and output channels will be grouped. Each group processed by different set of kernels.  
*grouped convolution (right)* with its equivalent with parallel convolution pipelines (left):

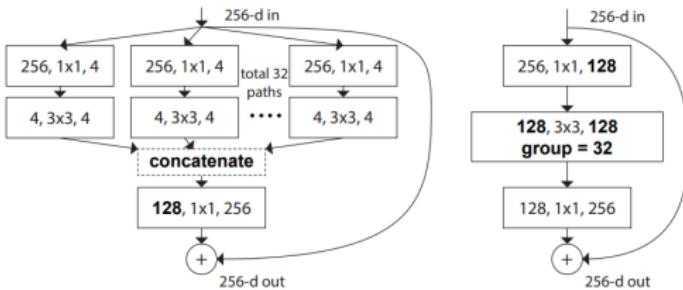


Figure from [Xie & Girshick<sup>+</sup> 17]. Idea used already in [Krizhevsky & Sutskever<sup>+</sup> 12].

- dilation: spacing between kernel elements. For *dilated convolution*. See [https://github.com/vdumoulin/conv\\_arithmetic/blob/master/gif/dilation.gif](https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/dilation.gif). Got popular for audio processing [Oord & Dieleman<sup>+</sup> 16].

# Pooling layer

- **input:** image  $x$  with shape  $(C, H, W)$ .
- **output:** also image  $y$  with shape  $(C', H', W')$ . But smaller than  $x$ .
- Operation: take max/mean over small, local sliding window to reduce image resolution (**downsampling**).
- **Allows to reduce computation for the following layer.**
- **No model parameter.** Just a max/mean operation.
- Hyper-parameter: as for convolution, need to specify pooling window dimensions, and how to move the window (stride).

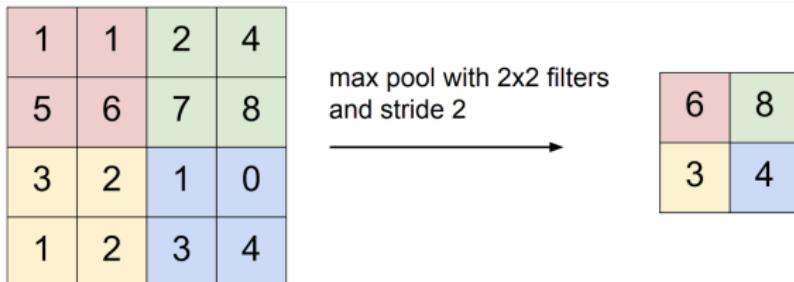


Figure taken from [Li & Johnson<sup>+</sup> 19a].

# PyTorch examples

- nn.Conv2d layer:  
→ Check input/output **shapes!**

```
1 >>> input = torch.randn(16, 3, 48, 48) # (B, C, H, W)
2 >>> layer = nn.Conv2d(3, 32, 3) # default padding=0
3 >>> layer.weight.shape
4 torch.Size([32, 3, 3, 3])
5 >>> layer.bias.shape
6 torch.Size([32])
7 >>> output = layer(input)
8 >>> output.size()
9 torch.Size([16, 32, 46, 46])
10 >>> layer = nn.Conv2d(3, 32, 3, padding=1)
11 >>> output = layer(input)
12 >>> output.size()
13 torch.Size([16, 32, 48, 48])
14 >>> # it's flexible:
15 >>> layer = nn.Conv2d(3, 32, (3, 5), stride=(2, 1), padding=(1, 2))
16 >>> output = layer(input)
17 >>> output.size()
18 torch.Size([16, 32, 24, 48])
```

# PyTorch examples (cont'd)

## ■ Max pooling layer: `nn.MaxPool2d`

```
1 >>> input = torch.randn(16, 3, 48, 48) # (B, C, H, W)
2 >>> pooling = nn.MaxPool2d(3)
3 >>> # non-overlapping pooling w/ window size (3, 3)
4 >>> output = pooling(input)
5 >>> output.size() # 48 / 3 = 16
6 torch.Size([16, 3, 16, 16])
7 >>> # overlapping pooling w/ stride (2, 2)
8 >>> pooling = nn.MaxPool2d(3, stride=2)
9 >>> output = pooling(input)
10 >>> output.size()
11 torch.Size([16, 3, 23, 23])
12 >>> pooling = nn.MaxPool2d(3, stride=2, padding=1)
13 >>> # same as stride=(2, 2), padding=(1, 1)
14 >>> output = pooling(input)
15 >>> output.size()
16 torch.Size([16, 3, 24, 24])
17 >>> # downsampling by a factor 2 using a window size 3.
```

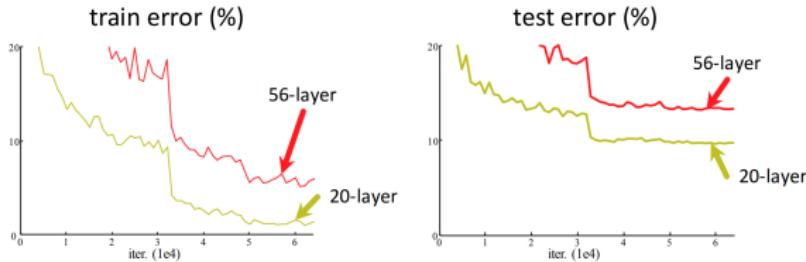
# Put them together

ConvNets are obtained by alternating convolutional and pooling layers:

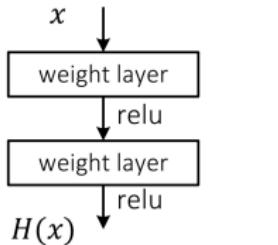
```
1 class ConvNet(nn.Module):
2     def __init__(self): # just example.
3         super(ConvNet, self).__init__()
4         self.conv1 = nn.Conv2d(3, 6, 5) # input shape (3, 32, 32).
5         self.pool = nn.MaxPool2d(2, 2)
6         self.conv2 = nn.Conv2d(6, 16, 5)
7         self.fc1 = nn.Linear(16 * 5 * 5, 128)
8         self.fc2 = nn.Linear(128, 64)
9         self.fc3 = nn.Linear(64, 10) # 10 output classes.
10
11    def forward(self, x):
12        x = self.pool(F.relu(self.conv1(x))) # conv, pool.
13        x = self.pool(F.relu(self.conv2(x))) # conv, pool.
14        x = x.view(-1, 16 * 5 * 5) # linearize input "images".
15        x = F.relu(self.fc1(x)) # fully connected.
16        x = F.relu(self.fc2(x)) # fully connected.
17        x = self.fc3(x) # fully connected.
18
19        return x
```

# So, shall we put more layers?

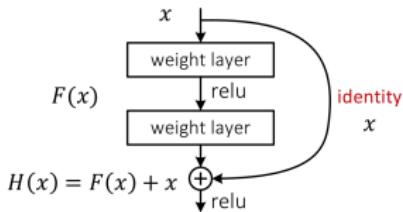
- Observation (all figures from [He & Zhang<sup>+</sup> 16c]):



- More layers should never hurt? If they learn identity in the worst case!



Standard network.



Residual block.

- Alternative intuition: simply inspired by LSTM-RNN (up next!).

# Very Deep NNs with skip connections

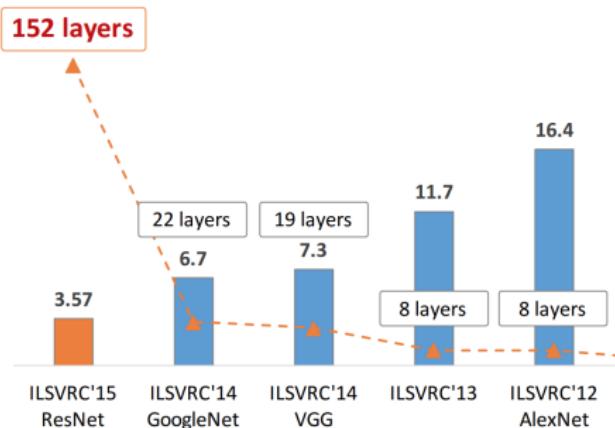
- **Skip connections:** if a layer applies transformation  $F$  to  $x$  to get  $y = F(x)$ , directly “connect”  $x$  to  $y$  by skipping the transformation.
- Two types:
  - **Highway connection/networks** [Srivastava & Greff<sup>+</sup> 15] from **IDSIA**.  
Gated connection like in LSTM (next subsection!):  
$$y = F(x) \odot g(x) + x \odot h(x),$$
where  $\odot$  denotes element-wise multiplication and the *gates*  $g$  and  $h$  are neural networks.
  - **Residual connection/networks** [He & Zhang<sup>+</sup> 16a].  
Simple addition:  $y = F(x) + x$



- Enable training models with more than 1000 layers in image recognition.
- Also used in Transformer architectures (see later...)

# Skip connections, performance

- ImageNet Large Scale Visual Recognition Competition (ILSVRC) top-5 error (%) (from [He & Zhang<sup>+</sup> 16c]):



- Many follow up works...
  - By the same authors: improved resblocks [He & Zhang<sup>+</sup> 16b] (deeper)
  - Wide residual nets [Zagoruyko & Komodakis 16]
  - DenseNets [Huang & Liu<sup>+</sup> 17, Lang & Witbrock 88]...

# Convolutional neural networks, beyond images

## Applications beyond image processing:

- Natural language: e.g.
  - Text classification [Kim 14]
  - Machine translation [Gehring & Auli<sup>+</sup> 17]
- Speech recognition [Waibel & Hanazawa<sup>+</sup> 89]

... and much more. We could have done the whole lecture on ConvNets...

## From engineering view point,

- Like pooling layers, convolutional layers (with stride  $\geq 2$ ) allow us to downsample the input (reduce input resolution).
- It's a general tool for trainable downsampling (e.g. reduce a long sequence to a shorter one).
- Convolution can be learned vs. max-pooling is parameter-free and cheap.

Interested in historical background? Watch Prof. Schmidhuber's video:

<https://youtu.be/ys0w6lNWx2o?t=20>

# Exercise 5 / Assignment 2

A few words on **Assignment 2**:

- You will be building a full pipeline for image classification using
  - convolutional neural networks
  - some practical training tricks
- Remember the usual workflow (Chapter 1)!
- Similar task in Exercise 4

# Outline

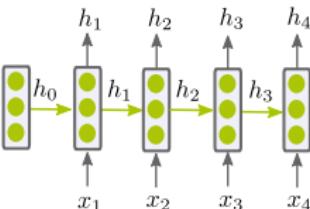
1. Workflow Overview
2. Introduction to Tools
- 3. Fundamental building blocks**
  1. Feed-forward NNs, Convolutional NNs, Residual NNs
  - 2. Recurrent NNs and LSTM**
  3. Attention, Self-attention, Transformers
4. Building models
5. Practical tricks and methods for training
6. Final words and Outlook

- A neural network architecture for processing **sequences**.
    - Consider a sequence of vectors  $(x_1, x_2, \dots, x_T)$  with  $x_t \in \mathbb{R}^D$
    - **At each time step**, standard RNNs update its **hidden state**  $h_t \in \mathbb{R}^H$  as a function of the **new input**  $x_t$  and the **previous hidden states**  $h_{t-1}$ :

$$h_t = f(Wx_t + Rh_{t-1} + b)$$

where

- $f$  is an activation function (e.g. tanh)
  - $W \in \mathbb{R}^{H \times D}$  and  $R \in \mathbb{R}^{H \times H}$  are weight matrices, and  $b \in \mathbb{R}^H$  a bias vector.



- Initial state  $h_0$ : typically chosen to be zero.
  - Compression of **variable length context**  $x_1, \dots, x_t$  to a fixed size vector  $h_t$ .

## Recurrent neural networks (cont'd)

- **Conceptually:** two inputs, two linear transformations, then sum the results.

$$h_t = f(Wx_t + Rh_{t-1} + b)$$

- Equivalent but better for efficient computation: **concatenate** the inputs, do **one** "big" linear transformation.

$$h_t = f([W, R] \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} + b)$$

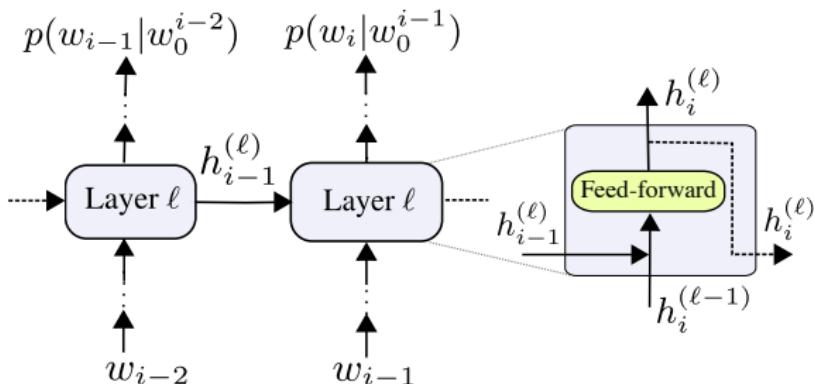
where  $[WR] \in \mathbb{R}^{H \times (D+H)}$  and  $\begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} \in \mathbb{R}^{(D+H) \times 1}$

- Look at the equation: it is just like the standard feedforward layer but with the previous output as a part of the input: recurrent.
- (You typically do not need to do this concatenation explicitly; PyTorch takes care of it internally)

# Recurrent neural networks, example

## Language modeling:

- **Task:** given a word sequence  $w_0, \dots, w_{i-1}$ , predict the next word  $w_i$ .
- **input** to the network at each time step: previous word  $w_{i-1} \in V$ , where  $V$  is the vocabulary.
- **output:** probability distribution over the vocabulary  $w \in V$ :  $p(w|w_0^{i-1})$ .  
(normalized vector of size  $|V|$ ). NB: notation  $w_0^{i-1} = (w_0, \dots, w_{i-1})$
- RNN state  $h_{i-1}$  compactly represents all previous words  $w_0^{i-1}$ .
- Step-by-step computation:



# Embedding layer

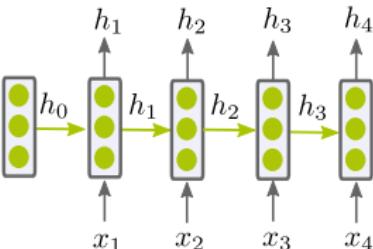
- Neural networks process **vectors**.
- Discrete input symbols (e.g. words) can be represented by **one-hot** vectors.
  - The size of a one-hot vector is the vocabulary size.
  - An ID is given to each word.
  - One hot vector's entries are all 0 except at the position corresponding to its ID where it's 1.

$$\begin{array}{rcl}
 0 & = & \begin{array}{|c|} \hline 1 \\ 0 \\ 0 \\ 0 \\ \hline \end{array} \\
 \text{"apple"} & & 
 \end{array}
 \quad
 \begin{array}{rcl}
 1 & = & \begin{array}{|c|} \hline 0 \\ 1 \\ 0 \\ 0 \\ \hline \end{array} \\
 \text{"banana"} & & 
 \end{array}
 \quad
 \begin{array}{rcl}
 2 & = & \begin{array}{|c|} \hline 0 \\ 0 \\ 1 \\ 0 \\ \hline \end{array} \\
 \text{"dog"} & & 
 \end{array}
 \quad
 \begin{array}{rcl}
 3 & = & \begin{array}{|c|} \hline 0 \\ 0 \\ 0 \\ 1 \\ \hline \end{array} \\
 \text{"car"} & & 
 \end{array}$$

- Matrix multiplication with a one-hot vector is a (column) **look-up** operation (try to write it down!).
- No need to do the actual multiplication.
- **Embedding layer**: input = discrete ID, output: its vector representation (trainable parameters).
  - Learning of the continuous representation (vector) of discrete tokens is part of the model.
  - In PyTorch: `nn.Embedding`

# Recurrent neural networks, training

- The most popular training algorithm: back-propagation *through time*.
- By unfolding the recurrence, we obtain a deep feed-forward neural network (see how many layers are between  $x_1$  and  $h_4$  in the figure)
  - You can apply back-propagation to the resulting network.
  - One update of parameters given a sequence (a batch of sequences).

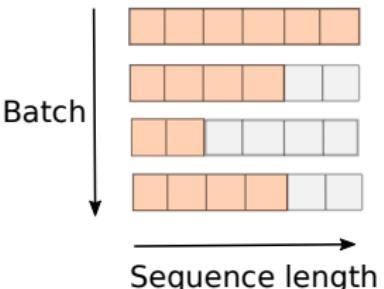


- Unfolded model is as deep as the number of time steps. Typical problems:
  - Exploding gradient
  - Vanishing gradient
- Exploding gradient can be alleviated by *clipping* gradient at some threshold (a hyper-parameter).

# Training, Batch, Padding

- How to create mini-batches to train RNNs?
- Batch of shape e.g. (sequence length, batch size , feature dimension)
- Sequences can have **variable lengths!**

We need to introduce **padding**:

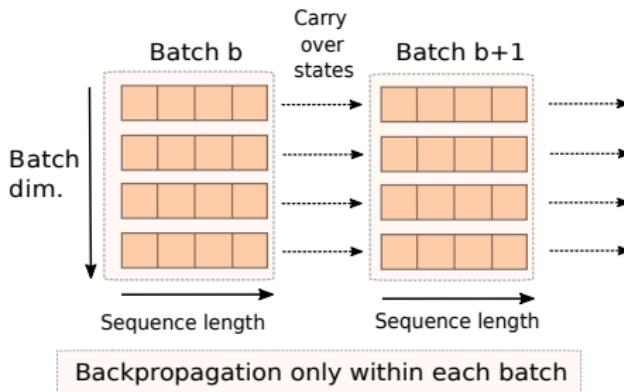


- Add *padding tokens* for short sequences such that all sequences get the same length (gray boxes above).
- Run the RNN on the padded batch.
- **Exclude the loss values from the padded positions.**

See for example `ignore_index` argument in `nn.CrossEntropyLoss`.

# Truncated back-propagation through time

- In some cases, sequences are too long for back-propagation through the entire sequence.
- We typically use **truncated** back-propagation through time.
  - Create fixed-length chunk/segments from the original sequences by splitting/concatenating them.
  - **Carry over** the state vectors between two consecutive batches in the forward pass, but
  - Only propagate gradients **within** the batch in the backward pass.



# Other architectures, long short-term memory (LSTM) and gating

Now look at this RNN:

- Three gates, input/forget/output gates (parameterized by an NN):

$$i_t = \sigma(W_i x_t + R_i h_{t-1} + b_i)$$

$$f_t = \sigma(W_f x_t + R_f h_{t-1} + b_f)$$

$$o_t = \sigma(W_o x_t + R_o h_{t-1} + b_o)$$

- Input candidate:

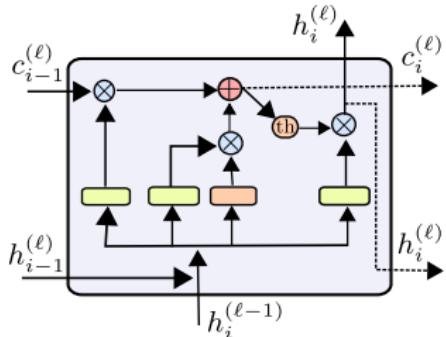
$$z_t = \tanh(W_z x_t + R_z h_{t-1} + b_z)$$

- Update cell states using the input and forget gates:

$$c_t = f_t \odot c_{t-1} + i_t \odot z_t$$

- Output:  $h_t = o_t \odot \tanh(c_t)$

(layer index  $(\ell)$  omitted in the equations.)

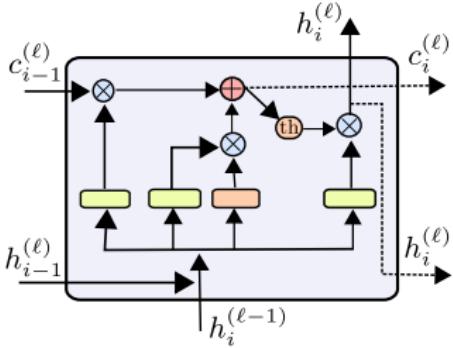


# Long short-term memory and gating (cont'd)

Special RNN architecture to alleviate vanishing gradient [Hochreiter & Schmidhuber 97, Gers & Schmidhuber<sup>+</sup> 00].

- two hidden state vectors
- internal cell state with an additive connection over time (no multiplication with a weight matrix! good gradient flow).
- **differentiable/soft multiplicative gates** (input/forget/output gates) control information flow around the cell
- each gate parameterized by a neural network.

Inspired many other new architectures!



# Long short-term memory and gating (cont'd)

- Again, linear transformations can be grouped.

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ z_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} \begin{bmatrix} W_i, R_i \\ W_f, R_f \\ W_o, R_o \\ W_z, R_z \end{bmatrix} \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} + \begin{bmatrix} b_i \\ b_f \\ b_o \\ b_z \end{bmatrix}$$

- The rest as in the previous slide:

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot z_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

NB:  $\begin{bmatrix} W_i, R_i \\ W_f, R_f \\ W_o, R_o \\ W_z, R_z \end{bmatrix} \in \mathbb{R}^{4H \times (D+H)}$  and  $\begin{bmatrix} b_i \\ b_f \\ b_o \\ b_z \end{bmatrix} \in \mathbb{R}^{4H \times 1}$

# Recurrent neural networks, implementations

Two types of RNN implementations:

■ Step-by-step RNN functions:

- take one input, output one output.
- expect you to write the loop over sequence.
- e.g. `torch.nn.LSTMCell`

■ Entire sequence RNN functions:

- take sequence, output sequence
- e.g. `torch.nn.LSTM`

- In general: you should use the entire sequence one whenever you can, which is optimized/faster.
  - High-level spirit: use built-in code as much as possible (avoid your own plain code in Python).

# Recurrent neural networks, implementations (cont'd)

Sequence-level function:

```
1 >>> rnn = nn.LSTM(10, 20, 2) # in_dim, out_dim, num_layers
2 >>> inputs = torch.randn(6, 3, 10) # (len, B, in_dim)
3 >>> h0 = torch.randn(2, 3, 20) # (num_layers, B, out_dim)
4 >>> c0 = torch.randn(2, 3, 20) # (num_layers, B, out_dim)
5 # outputs of shape (len, B, out_dim)
6 >>> outputs, (hn, cn) = rnn(inputs, (h0, c0))
7 >>> outputs.size()
8 torch.Size([6, 3, 20])
```

NB:

- Only possible if all inputs are known (for example in training, or in some evaluation setups).
- If the input also depends on the previous time steps (for example during *search*; we will see later) we have no other choice but to go step by step.

# Recurrent neural networks, implementations (cont'd)

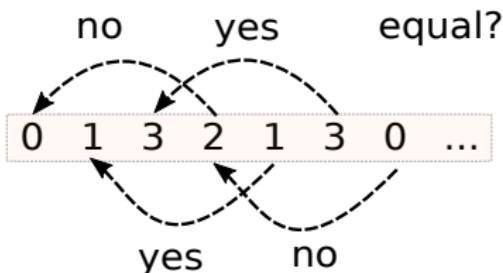
Step-by-step function:

```
1  >>> rnn = nn.LSTMCell(10, 20) # in_dim, out_dim
2  >>> input = torch.randn(6, 3, 10) # (len, B, in_dim)
3  >>> h = torch.randn(3, 20) # (B, out_dim)
4  >>> c = torch.randn(3, 20) # (B, out_dim)
5  >>> output = []
6  >>> for i in range(6): # "manual" Python loop
7      h, c = rnn(input[i], (h, c))
8      output.append(h)
9  >>> output = torch.stack(output, 0) # from list to tensor
10 >>> output.size()
11 torch.Size([6, 3, 20])
```

# Example toy task: $N$ -back

Task:

- Input: sequence of numbers (between 0 and  $k - 1$ ).
- Target at each position: 1 if the current input is equal to the number at the  $n$ -th position back. 0 otherwise.
- Sequences can have different lengths.



## Example toy task: $N$ -back, model

It is a binary classification at each position over a sequence.

Model:

- Read the input sequences using RNN.
- Input: number between 0 and  $k \rightarrow$  discrete!
- Actual input to the model: one hot representation: vector of size  $k$  with zero entry everywhere except the position (embedding layer is not needed though:  $k$  is small)
- A classifier linear layer which maps the RNN's hidden vector to the two output nodes (2 classes here; strictly speaking, one node is enough).
- Cross entropy loss for training.

# N-back, data generation

## ■ A helper function first:

```
1 def nback(n, k, length, random_state):
2     xi = random_state.randint(k, size=length)
3     yi = np.zeros(length, dtype=int)
4
5     for t in range(n, length):
6         yi[t] = (xi[t-n] == xi[t])
7
8     return xi, yi
```

# N-back, data generation (cont'd)

## ■ Main data generator:

```
1 def create_dataset_nback(n_sequences, mean_length, std_length,
2                           n, k, random_state):
3     X, Y, lengths = [], [], []
4     for _ in range(n_sequences):
5         length = random_state.normal(loc=mean_length, scale=std_length)
6         length = int(max(n+1, length))
7         xi, yi = nback(n, k, length, random_state)
8         X.append(xi)
9         Y.append(yi)
10        lengths.append(length)
11    max_len = max(lengths)
12
13    # We pad X w/ 0 (for one-hot), and Y w/ -1 (CE loss).
14    X_arr = np.zeros((n_sequences, max_len), dtype=np.int64)
15    Y_arr = np.zeros((n_sequences, max_len), dtype=np.int64) - 1
16
17    for i in range(n_sequences):
18        X_arr[i, 0: lengths[i]] = X[i]
19        Y_arr[i, 0: lengths[i]] = Y[i]
20
21    return X_arr, Y_arr, lengths
```

# N-back, data generation (cont'd)

## ■ Generate data:

```
1 import numpy as np
2
3 seed = 0
4 n = 3
5 k = 4
6 mean_length = 20
7 std_length = 5
8 n_sequences = 1000
9
10 random_state = np.random.RandomState(seed=seed)
11
12 X_train, Y_train, length_train = create_dataset_nback(
13     n_sequences, mean_length, std_length, n, k, random_state)
14
15 X_val, Y_val, length_val = create_dataset_nback(
16     n_sequences, mean_length, std_length, n, k, random_state)
```

- Create model: an RNN + a linear classifier layer.

```
1 import torch.nn as nn
2
3 input_dim = 4
4 hidden_size = 64
5 num_layers = 1
6 num_classes = 2
7
8 rnn = nn.RNN(input_dim, hidden_size, num_layers)
9 linear = nn.Linear(hidden_size, num_classes)
10 h0 = torch.zeros(num_layers, n_sequences, hidden_size) # init state

1 import torch.optim as optim
2 # Again: alternatively write a model class!
3 params = list(rnn.parameters()) + list(linear.parameters())
4 optimizer = optim.Adam(params, lr=0.01)
5
6 # Exclude padded position (position with -1).
7 loss_fn = nn.CrossEntropyLoss(ignore_index=-1)
```

# N-back, prepare data

- Prepare data in the form/shape expected by RNNs:

```
1 # prepare data
2 X = torch.from_numpy(X_train)
3 X = torch.nn.functional.one_hot(X)    # (B, len, k)
4 X = X.transpose(0, 1)
5 X = X.float()
6
7 y = torch.from_numpy(Y_train)
8 y = y.transpose(0, 1).flatten()
9
10 # same for val:
11 X_val = torch.from_numpy(X_val)
12 X_val = torch.nn.functional.one_hot(X_val)
13 X_val = X_val.transpose(0, 1).float()
14 y_val = torch.from_numpy(Y_val).transpose(0, 1).flatten()
```

# N-back, training

```
1 num_train_steps = 200
2
3 total_val = sum(length_val)
4 total_tr = sum(length_train)
5
6 for step in range(num_train_steps):
7     # more convenient if you had defined a model!
8     # Please fix this in the exercise 6!
9     rnn.train()
10    linear.train()
11
12    optimizer.zero_grad()
13    output, hn = rnn(X, h0)
14    output = output.view(-1, hidden_size) # (B*len, dim)
15    output = linear(output)
16
17    loss = loss_fn(output, y)
18    print(f"training loss: {loss}")
19
20    loss.backward()
21    optimizer.step()
22
23 # ... continue to the next slide ...
```

# N-back, training (cont'd)

```
1 # ... continue from the previous slide ...
2 rnn.eval()
3 linear.eval() # more convenient if you had defined a model!
4 with torch.no_grad():
5     # Here evaluate also on training set (exercise 6).
6
7     output_val, hn_val = rnn(X_val, h0)
8     output_val = output_val.view(-1, hidden_size)
9     output_val = linear(output_val)
10
11    _, predicted_val = outputs_val.max(dim=1)
12    correct_val = (predicted_val == y_val)
13
14    # Important: do not count padded position!!
15    mask_val = (y_val >= 0)
16    correct_val = (correct_val * mask_val).sum().item()
17
18    print(f'epoch: {step}, val acc: {100 * correct_val / total_val}')
```

## Preview:

# Exercise 6 & Exercise 7 / Assignment 3

Both **Exercise 6** and **Exercise 7 / Assignment 3**

- Introduction to language modeling with RNNs

# Preliminaries for Assignment 3, Reminders

- Language models compute  $p(w_i | w_0^{i-1})$

Notation:  $w_0^{i-1} = (w_0, w_1, \dots, w_{i-2}, w_{i-1})$

- Given a sentence  $w_1^N$ ,

$$p(w_1, \dots, w_N) = \prod_{i=1}^N p(w_i | w_0^{i-1}) \text{ where}$$

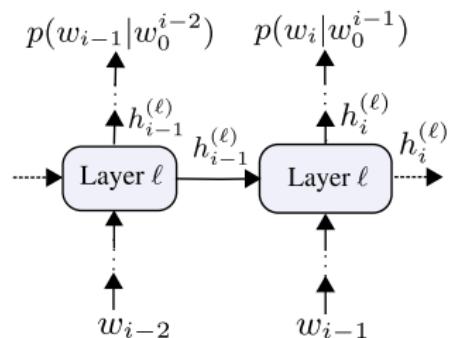
$w_0$  denotes an artificial start symbol

- This can be used to compute probabilities of sentences, e.g., which sentence should get a higher probability?

- $p(\text{I hate cars } <\text{eos}>)$
- $p(\text{I ate cars } <\text{eos}>)$

where  $<\text{eos}>$  is the end-of-sentence token.

- Another application in this assignment: text completion/generation.



# Preliminaries for Assignment 3

- This **assignment**: text generation/completion.
    - Once the model is trained, you will provide a beginning of some text  $w_0^{i-1}$ .
    - You let the model complete your text.
- You: Dogs like best to
- Your LM: eat , play , and sleep

# Preliminaries for Assignment 3

- This **assignment**: text generation/completion.
    - Once the model is trained, you will provide a beginning of some text  $w_0^{i-1}$ .
    - You let the model complete your text.
- You: Dogs like best to
- Your LM: eat , play , and sleep

How can this be done?

# Preliminaries for Assignment 3

- This **assignment**: text generation/completion.
    - Once the model is trained, you will provide a beginning of some text  $w_0^{i-1}$ .
    - You let the model complete your text.
- You: Dogs like best to  
Your LM: eat , play , and sleep

How can this be done?

- Given the beginning of a sentence  $w_0^{n-1}$ , let your LM compute  $p(.|w_0^{n-1})$  (i.e., the probability distribution over the **next** word/token)
- Then there are 2 possibilities:
  - (1) Take the word with the highest probability.  $\hat{w} = \operatorname{argmax}_w p(w|w_0^{n-1})$
  - (2) Randomly sample  $\hat{w}$  from the distribution  $p(w|w_0^{n-1})$
- Then feed the chosen word  $\hat{w}$  to the LM as an input, and continue for a fixed number of steps.

# Preliminaries for Assignment 3, Sampling vs. Search.

- **Search:** we want to find the most likely output from the model.
    - In the assignment, **greedy search:**
      - At each time step, obtain the most likely token.
  - **Sampling:** we want to generate diverse outputs from the model.
    - Randomly sample according to the model's output distribution.
- In both cases, use the corresponding model output token as the input to the model for the next time step.

# Preliminaries for Assignment 3, Pre-processing/Tokenization

The data is a plain text file.

- The modeling unit must be defined: the text must be **tokenized**, e.g.
  - Word level: i.e. split the text by white space.
  - Character level

Side note (not needed for A3): in general, depending on the task, we also have to do some text pre-processing:

- Normalization of lower/upper case, numbers, punctuations, or spelling, remove strings which are not really texts, etc.

**Vocabulary** of the model:

- Add all characters found in the training set.
- Add **special** tokens: pad token, unknown token, end-of-sentence token (not needed for A3), ...

See the helper code available on iCorsi.

# Outline

1. Workflow Overview
2. Introduction to Tools
- 3. Fundamental building blocks**
  1. Feed-forward NNs, Convolutional NNs, Residual NNs
  2. Recurrent NNs and LSTM
  3. Attention, Self-attention, Transformers
4. Building models
5. Practical tricks and methods for training
6. Final words and Outlook

# Illustration first!

```
1 >>> dict = { 'apple': 4, 'banana': 2, 'orange': 5} # fruit counts
2 >>> dict.keys()
3 dict_keys(['apple', 'banana', 'orange'])
4 >>> dict.values()
5 dict_values([4, 2, 5])
6 >>> dict['orange'] # query 'orange'
7 5
```

- A dictionary **stores key** and **value** pairs.
- Retrieval: the **query** (here orange) is **compared** to the keys, and the value corresponding to the matching key is returned.

# Illustration first!

```
1 >>> dict = { 'apple': 4, 'banana': 2, 'orange': 5} # fruit counts
2 >>> dict.keys()
3 dict_keys(['apple', 'banana', 'orange'])
4 >>> dict.values()
5 dict_values([4, 2, 5])
6 >>> dict['orange'] # query 'orange'
7 5
```

- A dictionary **stores key** and **value** pairs.
- Retrieval: the **query** (here orange) is **compared** to the keys, and the value corresponding to the matching key is returned.

**Can we do this in a differentiable way using neural networks?**

# Illustration first!

```

1 >>> dict = { 'apple': 4, 'banana': 2, 'orange': 5} # fruit counts
2 >>> dict.keys()
3 dict_keys(['apple', 'banana', 'orange'])
4 >>> dict.values()
5 dict_values([4, 2, 5])
6 >>> dict['orange'] # query 'orange'
7 5

```

- A dictionary **stores key** and **value** pairs.
- Retrieval: the **query** (here orange) is **compared** to the keys, and the value corresponding to the matching key is returned.

## Can we do this in a differentiable way using neural networks?

- **How to handle symbols?** Reminder from the last lecture: replace each symbol (key, query, value) by its vector representation.

# Illustration first!

```

1 >>> dict = { 'apple': 4, 'banana': 2, 'orange': 5} # fruit counts
2 >>> dict.keys()
3 dict_keys(['apple', 'banana', 'orange'])
4 >>> dict.values()
5 dict_values([4, 2, 5])
6 >>> dict['orange'] # query 'orange'
7 5
  
```

- A dictionary **stores key** and **value** pairs.
- Retrieval: the **query** (here orange) is **compared** to the keys, and the value corresponding to the matching key is returned.

## Can we do this in a differentiable way using neural networks?

- **How to handle symbols?** Reminder from the last lecture: replace each symbol (key, query, value) by its vector representation.
- **How to compare query and key?** E.g., compute the dot product between query and key vectors.

# Illustration first!

```

1 >>> dict = { 'apple': 4, 'banana': 2, 'orange': 5} # fruit counts
2 >>> dict.keys()
3 dict_keys(['apple', 'banana', 'orange'])
4 >>> dict.values()
5 dict_values([4, 2, 5])
6 >>> dict['orange'] # query 'orange'
7 5
  
```

- A dictionary **stores key** and **value** pairs.
- Retrieval: the **query** (here orange) is **compared** to the keys, and the value corresponding to the matching key is returned.

## Can we do this in a differentiable way using neural networks?

- **How to handle symbols?** Reminder from the last lecture: replace each symbol (key, query, value) by its vector representation.
- **How to compare query and key?** E.g., compute the dot product between query and key vectors.
- **How to output the value of the matching key (in a differentiable way)?**  
Return the *weighted sum* of all value vectors where weights are the key/query similarity scores.

# Attention with neural networks

- Two lists of  $N$  vectors each, with dimensions  $d_{\text{key}}$  and  $d_{\text{value}}$ 
  - **keys**  $\mathcal{K} = (k_1, \dots, k_N) \in \mathbb{R}^{d_{\text{key}} \times N}$
  - **values**  $\mathcal{V} = (v_1, \dots, v_N) \in \mathbb{R}^{d_{\text{value}} \times N}$
- One vector  $q \in \mathbb{R}^{d_{\text{key}} \times 1}$  (called **query**).

# Attention with neural networks

- Two lists of  $N$  vectors each, with dimensions  $d_{\text{key}}$  and  $d_{\text{value}}$ 
  - **keys**  $\mathcal{K} = (k_1, \dots, k_N) \in \mathbb{R}^{d_{\text{key}} \times N}$
  - **values**  $\mathcal{V} = (v_1, \dots, v_N) \in \mathbb{R}^{d_{\text{value}} \times N}$
- One vector  $q \in \mathbb{R}^{d_{\text{key}} \times 1}$  (called **query**).

Between each **key** vector  $k_i$  with  $1 \leq i \leq N$  and the **query** vector  $q$ , the *similarity score*  $s_i \in \mathbb{R}$  is computed as:

$$s_i = k_i \bullet q \quad \bullet \text{ denotes the dot product.}$$

# Attention with neural networks

- Two lists of  $N$  vectors each, with dimensions  $d_{\text{key}}$  and  $d_{\text{value}}$ 
  - **keys**  $\mathcal{K} = (k_1, \dots, k_N) \in \mathbb{R}^{d_{\text{key}} \times N}$
  - **values**  $\mathcal{V} = (v_1, \dots, v_N) \in \mathbb{R}^{d_{\text{value}} \times N}$
- One vector  $q \in \mathbb{R}^{d_{\text{key}} \times 1}$  (called **query**).

Between each **key** vector  $k_i$  with  $1 \leq i \leq N$  and the **query** vector  $q$ , the *similarity score*  $s_i \in \mathbb{R}$  is computed as:

$$s_i = k_i \bullet q \quad \bullet \text{ denotes the dot product.}$$

This yields a similarity score vector  $s \in \mathbb{R}^N$  that is then re-normalized to  $\alpha = (\alpha_1, \dots, \alpha_i, \dots, \alpha_N) \in \mathbb{R}^N$  by:

$$\alpha = \text{softmax}(s) \quad \text{where } s = (s_1, \dots, s_i, \dots, s_N) \in \mathbb{R}^N.$$

# Attention with neural networks

- Two lists of  $N$  vectors each, with dimensions  $d_{\text{key}}$  and  $d_{\text{value}}$ 
  - **keys**  $\mathcal{K} = (k_1, \dots, k_N) \in \mathbb{R}^{d_{\text{key}} \times N}$
  - **values**  $\mathcal{V} = (v_1, \dots, v_N) \in \mathbb{R}^{d_{\text{value}} \times N}$
- One vector  $q \in \mathbb{R}^{d_{\text{key}} \times 1}$  (called **query**).

Between each **key** vector  $k_i$  with  $1 \leq i \leq N$  and the **query** vector  $q$ , the *similarity score*  $s_i \in \mathbb{R}$  is computed as:

$$s_i = k_i \bullet q \quad \bullet \text{ denotes the dot product.}$$

This yields a similarity score vector  $s \in \mathbb{R}^N$  that is then re-normalized to  $\alpha = (\alpha_1, \dots, \alpha_i, \dots, \alpha_N) \in \mathbb{R}^N$  by:

$$\alpha = \text{softmax}(s) \quad \text{where } s = (s_1, \dots, s_i, \dots, s_N) \in \mathbb{R}^N.$$

These scores are used to compute the **weighted average** of **value** vectors  $v_i$ :

$$\text{Attention}(\mathcal{K}, \mathcal{V}, q) = \sum_{i=1}^N \alpha_i v_i.$$

# Attention with neural networks, comments

- These operations can be written as matrix operations:

$$\text{Attention}(\mathcal{K}, \mathcal{V}, q) = \mathcal{V} \text{softmax}(\mathcal{K}^\top q)$$

# Attention with neural networks, comments

- These operations can be written as matrix operations:

$$\text{Attention}(\mathcal{K}, \mathcal{V}, q) = \mathcal{V} \text{softmax}(\mathcal{K}^T q)$$

- A high attention score  $\alpha_i$  indicates the **attention** focused on the content  $(k_i, v_i)$  when the input is  $q$ , that is something we can visualize.

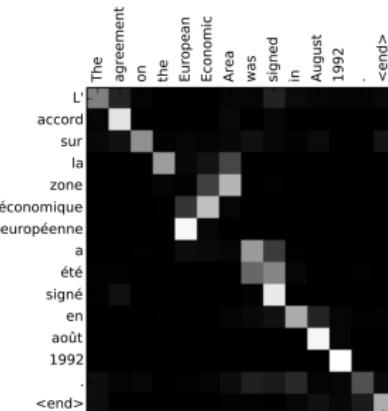


Figure taken from [\[Bahdanau & Cho<sup>+</sup> 15\]](#) for illustration.

We will see the English to French translation model in the next section.

# Attention computation

There are various ways to compute the similarity scores  $\text{sim}(q, k) \in \mathbb{R}$  between two vectors  $q, k \in \mathbb{R}^{d \times 1}$ :

- dot attention [Luong & Pham<sup>+</sup> 15]

$$\text{sim}(q, k) = q \bullet k = q^T k$$

# Attention computation

There are various ways to compute the similarity scores  $\text{sim}(q, k) \in \mathbb{R}$  between two vectors  $q, k \in \mathbb{R}^{d \times 1}$ :

- dot attention [Luong & Pham<sup>+</sup> 15]

$$\text{sim}(q, k) = q \bullet k = q^T k$$

- scaled dot attention [Vaswani & Shazeer<sup>+</sup> 17]

$$\text{sim}(q, k) = \frac{q^T k}{\sqrt{d}}$$

# Attention computation

There are various ways to compute the similarity scores  $\text{sim}(q, k) \in \mathbb{R}$  between two vectors  $q, k \in \mathbb{R}^{d \times 1}$ :

- dot attention [Luong & Pham<sup>+</sup> 15]

$$\text{sim}(q, k) = q \bullet k = q^T k$$

- scaled dot attention [Vaswani & Shazeer<sup>+</sup> 17]

$$\text{sim}(q, k) = \frac{q^T k}{\sqrt{d}}$$

( $\sqrt{d}$  because if elements in  $q$  and  $k$  are i.i.d random variables with mean 0 and variance 1,  $q^T k$  has mean 0 and variance  $d$ .)

# Attention computation

There are various ways to compute the similarity scores  $\text{sim}(q, k) \in \mathbb{R}$  between two vectors  $q, k \in \mathbb{R}^{d \times 1}$ :

- dot attention [Luong & Pham<sup>+</sup> 15]

$$\text{sim}(q, k) = q \bullet k = q^T k$$

- scaled dot attention [Vaswani & Shazeer<sup>+</sup> 17]

$$\text{sim}(q, k) = \frac{q^T k}{\sqrt{d}}$$

( $\sqrt{d}$  because if elements in  $q$  and  $k$  are i.i.d random variables with mean 0 and variance 1,  $q^T k$  has mean 0 and variance  $d$ .)

- MLP attention [Bahdanau & Cho<sup>+</sup> 15]

$$\text{sim}(q, k) = w^T \tanh(W[q, k] + b)$$

where  $W \in \mathbb{R}^{d \times 2d}$  and  $w, b \in \mathbb{R}^{d \times 1}$  are trainable parameters.

## Attention computation (cont'd)

In practice:

- **Scaled dot attention** works well while being efficient, and it has no parameter.
- **Multi-head attention** is often used:
  - Split each of key/value/query vectors into  $H$  sub-vectors.  $H$  is the number of heads.
  - Compute separate attention operations for each key/value/query sub-vectors.
  - Concatenate (feature dimension) the results from each head to get the final output.
- e.g., if the dimension is 512 (for each of key, value and query), and we use 8 attention **heads**, each attention computation is carried out for 64 dimensional sub-vectors.
- `torch.nn.MultiheadAttention`

# Attention with neural networks, applications

- Originally proposed for machine translation (next section).
- Intuition: focus only on parts of the input sentence, while producing parts of target sentence...
- Impact on many other applications: attention is everywhere now.
- **Differentiable implementation of dictionary/database retrieval.**
- Fundamental idea of **ignoring irrelevant information**.
- Visualization and interpretability.

# Self-attention (autoregressive version)

Can we build a general purpose sequence processing layer based on attention?

- Alternative to RNNs to process sequences.

# Self-attention (autoregressive version)

Can we build a general purpose sequence processing layer based on attention?

- Alternative to RNNs to process sequences.
- Note: autoregressive means that we process a sequence  $x_1^N$  from left to right (or right to left) step by step: i.e. while predicting a token  $x_n$ , we make use of information from the past  $x_1^{n-1}$ .

# Self-attention (autoregressive version)

Can we build a general purpose sequence processing layer based on attention?

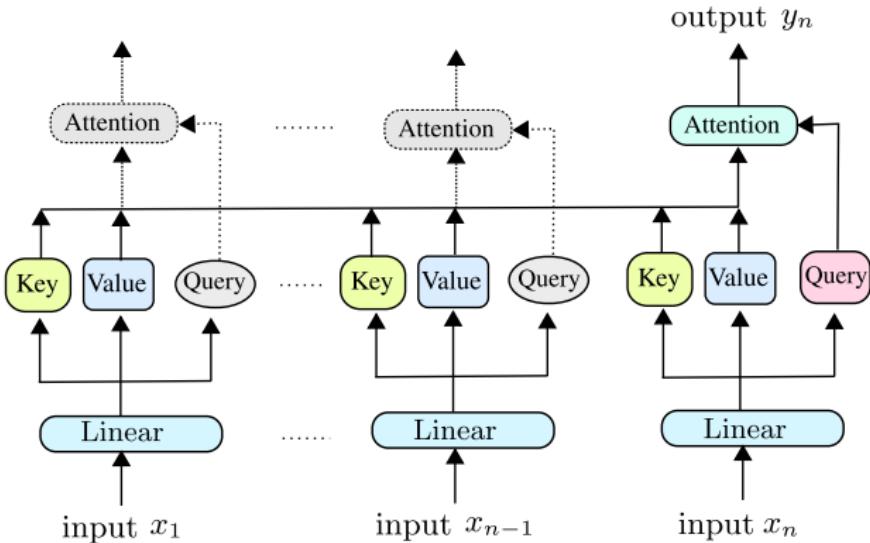
- Alternative to RNNs to process sequences.
- Note: autoregressive means that we process a sequence  $x_1^N$  from left to right (or right to left) step by step: i.e. while predicting a token  $x_n$ , we make use of information from the past  $x_1^{n-1}$ .
- Basic idea: “enlarge the database for each new input”
  - Each input transformed in 3 ways: key, value, and query vectors.
  - Store key and value vectors from all predecessor inputs (“database”)
  - Compute the output via attention using the query vector.

# Self-attention (autoregressive version)

Can we build a general purpose sequence processing layer based on attention?

- Alternative to RNNs to process sequences.
- Note: autoregressive means that we process a sequence  $x_1^N$  from left to right (or right to left) step by step: i.e. while predicting a token  $x_n$ , we make use of information from the past  $x_1^{n-1}$ .
- Basic idea: “enlarge the database for each new input”
  - Each input transformed in 3 ways: key, value, and query vectors.
  - Store key and value vectors from all predecessor inputs (“database”)
  - Compute the output via attention using the query vector.
- Does this make sense?

# Self-attention, illustration



- **Abandon compression** ability of RNNs!
- Perform very well as a part of **Transformer** models (up next).

# Self-attention (cont'd)

- Consider a sequence of vectors  $(x_1, \dots, x_N)$ :

## Self-attention (cont'd)

- Consider a sequence of vectors  $(x_1, \dots, x_N)$ :
- At step  $n$ , input  $x_n \in \mathbb{R}^{D \times 1}$  is first projected into **three** vectors:

$$q_n, k_n, v_n = Qx_n, Kx_n, Vx_n \quad Q, K \in \mathbb{R}^{d_{\text{key}} \times D}, V \in \mathbb{R}^{d_{\text{value}} \times D}$$

$Q, K, V$  are the **trainable parameters** of the layer.

## Self-attention (cont'd)

- Consider a sequence of vectors  $(x_1, \dots, x_N)$ :
- At step  $n$ , input  $x_n \in \mathbb{R}^{D \times 1}$  is first projected into **three** vectors:

$$q_n, k_n, v_n = Qx_n, Kx_n, Vx_n \quad Q, K \in \mathbb{R}^{d_{\text{key}} \times D}, V \in \mathbb{R}^{d_{\text{value}} \times D}$$

- $Q, K, V$  are the **trainable parameters** of the layer.
- The **memory** of the layer consists of **concatenation** of  $k_i$  and  $v_i$  vectors (along position/time dimension) for all predecessor inputs. Thus, at step  $n$ :

$$\begin{aligned}\mathcal{K}_n &= \text{Concat}(\mathcal{K}_{n-1}, k_n) \in \mathbb{R}^{d \times n} \\ \mathcal{V}_n &= \text{Concat}(\mathcal{V}_{n-1}, v_n) \in \mathbb{R}^{d \times n}\end{aligned}$$

## Self-attention (cont'd)

- Consider a sequence of vectors  $(x_1, \dots, x_N)$ :
- At step  $n$ , input  $x_n \in \mathbb{R}^{D \times 1}$  is first projected into **three** vectors:

$$q_n, k_n, v_n = Qx_n, Kx_n, Vx_n \quad Q, K \in \mathbb{R}^{d_{\text{key}} \times D}, V \in \mathbb{R}^{d_{\text{value}} \times D}$$

- $Q, K, V$  are the **trainable parameters** of the layer.
- The **memory** of the layer consists of **concatenation** of  $k_i$  and  $v_i$  vectors (along position/time dimension) for all predecessor inputs. Thus, at step  $n$ :

$$\begin{aligned}\mathcal{K}_n &= \text{Concat}(\mathcal{K}_{n-1}, k_n) \in \mathbb{R}^{d \times n} \\ \mathcal{V}_n &= \text{Concat}(\mathcal{V}_{n-1}, v_n) \in \mathbb{R}^{d \times n}\end{aligned}$$

State size  $h_n = (\mathcal{K}_n, \mathcal{V}_n)$  increases with the sequence length; unlike in RNNs!

## Self-attention (cont'd)

- Consider a sequence of vectors  $(x_1, \dots, x_N)$ :
- At step  $n$ , input  $x_n \in \mathbb{R}^{D \times 1}$  is first projected into **three** vectors:

$$q_n, k_n, v_n = Qx_n, Kx_n, Vx_n \quad Q, K \in \mathbb{R}^{d_{\text{key}} \times D}, V \in \mathbb{R}^{d_{\text{value}} \times D}$$

- $Q, K, V$  are the **trainable parameters** of the layer.
- The **memory** of the layer consists of **concatenation** of  $k_i$  and  $v_i$  vectors (along position/time dimension) for all predecessor inputs. Thus, at step  $n$ :

$$\begin{aligned}\mathcal{K}_n &= \text{Concat}(\mathcal{K}_{n-1}, k_n) \in \mathbb{R}^{d \times n} \\ \mathcal{V}_n &= \text{Concat}(\mathcal{V}_{n-1}, v_n) \in \mathbb{R}^{d \times n}\end{aligned}$$

State size  $h_n = (\mathcal{K}_n, \mathcal{V}_n)$  increases with the sequence length; unlike in RNNs!

- Output  $y_n$  is computed by **attention** using these vectors:

$$y_n = \text{SelfAttention}(h_{n-1}, x_n) = \text{Attention}(\mathcal{K}_n, \mathcal{V}_n, q_n)$$

# Self-attention, positional encoding

- One more concept needs to be introduced: **positional encoding**.
- Attention operation is invariant to shuffling key-value pairs  $\{(k_i, v_i)\}_{1 \leq i \leq N}$ .
  - Positional information needs to be provided explicitly.
  - Positional encoding is a vector of the same size as input word/token embeddings, representing the position.
- We **add** the positional encoding vectors to the input embeddings.

# Self-attention, positional encoding (cont'd)

- Common model: **sinusoidal positional encoding**.

$e(i) \in \mathbb{R}^D$  representing the position  $i \in \mathbb{N}$ . Each component of  $e(i)$  is computed as follows: for  $0 \leq k < \frac{D}{2}$

$$e(i)_{2k} = \sin(i/10000^{2k/D})$$
$$e(i)_{2k+1} = \cos(i/10000^{2k/D})$$

# Self-attention, positional encoding (cont'd)

- Common model: **sinusoidal positional encoding**.

$e(i) \in \mathbb{R}^D$  representing the position  $i \in \mathbb{N}$ . Each component of  $e(i)$  is computed as follows: for  $0 \leq k < \frac{D}{2}$

$$e(i)_{2k} = \sin(i/10000^{2k/D})$$

$$e(i)_{2k+1} = \cos(i/10000^{2k/D})$$

Original motivation:  $e(i + m)$  linearly depends on  $e(i)$   
 (effective benefit of this property is not fully known).  
 The choice of "10000": a *large* number.

# Self-attention, positional encoding (cont'd)

- Common model: **sinusoidal positional encoding**.

$e(i) \in \mathbb{R}^D$  representing the position  $i \in \mathbb{N}$ . Each component of  $e(i)$  is computed as follows: for  $0 \leq k < \frac{D}{2}$

$$e(i)_{2k} = \sin(i/10000^{2k/D})$$

$$e(i)_{2k+1} = \cos(i/10000^{2k/D})$$

Original motivation:  $e(i + m)$  linearly depends on  $e(i)$  (effective benefit of this property is not fully known).

The choice of "10000": a *large* number.

- Standard approach: this vector is added (element-wise) to the input token/word embedding vector.

## Self-attention, positional encoding (cont'd)

- Common model: **sinusoidal positional encoding**.

$e(i) \in \mathbb{R}^D$  representing the position  $i \in \mathbb{N}$ . Each component of  $e(i)$  is computed as follows: for  $0 \leq k < \frac{D}{2}$

$$e(i)_{2k} = \sin(i/10000^{2k/D})$$

$$e(i)_{2k+1} = \cos(i/10000^{2k/D})$$

Original motivation:  $e(i + m)$  linearly depends on  $e(i)$  (effective benefit of this property is not fully known).

The choice of "10000": a *large* number.

- Standard approach: this vector is added (element-wise) to the input token/word embedding vector.
- Studying and improving positional encoding has been also a common research topic for self-attention based models.

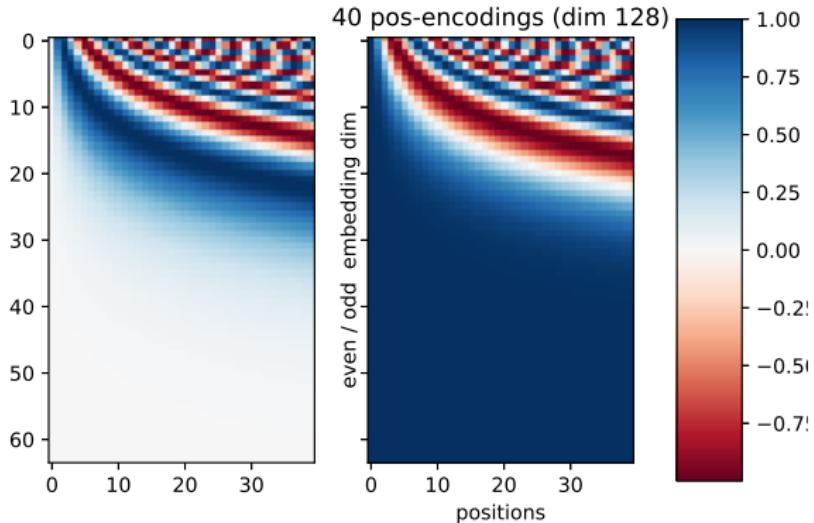
# Positional encoding, implementation

```
1 class PositionalEncoding(nn.Module):
2     """Example adapted from:
3
4         https://pytorch.org/tutorials/beginner/transformer_tutorial.html
5     """
6     def __init__(self, d_model, max_len=5000):
7         super(PositionalEncoding, self).__init__()
8         self.max_len = max_len
9
10    pe = torch.zeros(max_len, d_model)
11    position = torch.arange(
12        0, max_len, dtype=torch.float).unsqueeze(1)
13    div_term = torch.exp(torch.arange(0, d_model, 2).float()
14                         * (-math.log(10000.0) / d_model))
15    pe[:, 0::2] = torch.sin(position * div_term)
16    pe[:, 1::2] = torch.cos(position * div_term)
17
18    # shape (max_len, 1, dim)
19    pe = pe.unsqueeze(0).transpose(0, 1)
20    self.register_buffer('pe', pe) # Will not be trained.
21
22    # ... continue to next slide ...
```

# Positional encoding, implementation (cont'd)

```
1 # ... continue from previous slide ...
2
3     def forward(self, x):
4         # shape of x: (len, B, dim)
5         assert x.size(0) < self.max_len, (
6             f"Too long sequence: increase 'max_len'")
7         # shape of x (len, B, dim)
8         x = x + self.pe[:x.size(0), :]
9         return x
```

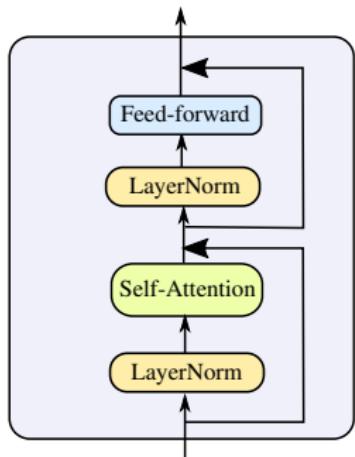
# Positional encoding, visualization



- Left: even, right: odd coordinates.
- Positional information is encoded in a few coordinates:
  - e.g., compare the vectors representing position 0 vs. position 30.
- This will be added to the token embedding vector.

# Transformer (auto-regressive version)

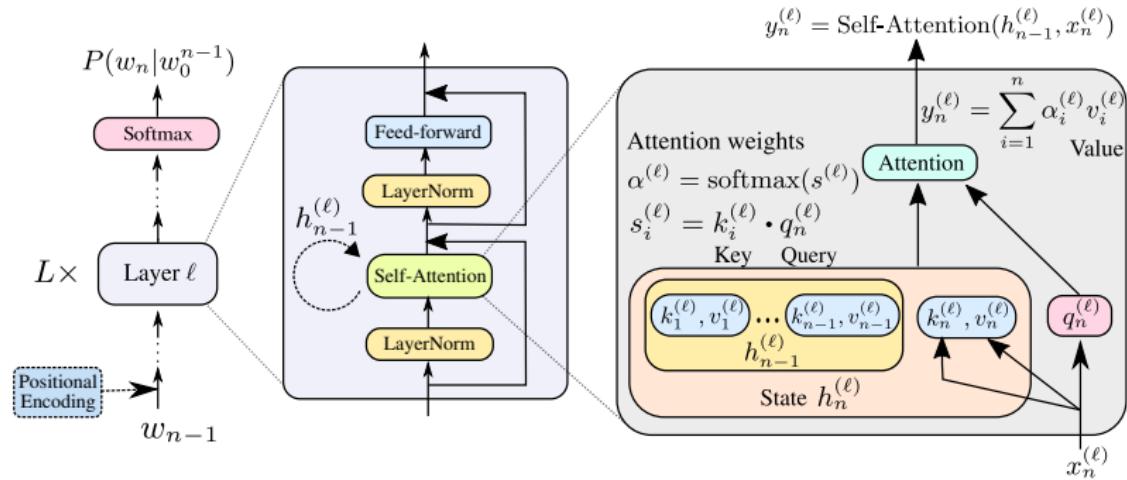
- One Transformer layer [Vaswani & Shazeer<sup>+</sup> 17] consists of multiple sub-layers:
  - One **self-attention layer**
  - One **feed-forward layer**
  - (We'll also add one *cross attention layer* in the sequence-to-sequence version that we'll see next week!).
- with helper components which facilitate training:
  - **Residual connection**  
(which we have seen in the last section)
  - **Layer normalization**  
(mean/variance normalization on feature dimension)



Typical Transformer models has multiple layers (up to 100; depending on the problem).

# Transformer layer, illustration (autoregressive case)

Illustration with a Transformer language model  
(as a generic example for sequence processing):



Many impactful models, e.g. OpenAI's GPT-2 & 3 models.

# Transformer (cont'd)

- Made self-attention very popular.
- Latest largest advancement in neural network architecture (2017).
- You will be likely using a Transformer layer instead of self-attention layer alone.
- `torch.nn.Transformer` : we will see this in details in the next chapter
- The **final assignment** is about Transformers.

# Summary

## What have we learned?

- Different neural network architectures for different problems.

Key words:

- Convolutional neural networks
- Recurrent neural networks
- Long short-term memory
- Attention

- Intuitions reflected in the design of these models.

## Coming up next...

- How to build complete model for different problems using these building blocks (next Chapter)

# Outline

1. Workflow Overview
2. Introduction to Tools
3. Fundamental building blocks
- 4. Building models**
5. Practical tricks and methods for training
6. Final words and Outlook

# Time to assemble pieces!

- In Chapter 3, we learned fundamental building blocks.
- In principle: we can now build neural network based models for MANY problems involving image-like data (2D structured data) and natural language-like data (sequences).



Figures adapted from [Le & Venkatesh 20] for illustration.

# Haven't we already built models?

- Yes. For some tasks, building models was straightforward.  
Example:
  - Basic image classification models (assignment 2): convolutional layers, pooling layers, fully connected layers, output classifier layer (softmax).
  - Basic language models (assignment 3): recurrent neural networks, input embedding layer, output classification layer.
- But you can do much more using what we learned in Chapter 3.
  - E.g. sequence to sequence problems.

# Preliminary comments

- We have input(s) and output(s) defined by the task.
- **Basic idea:** We parameterize the input-to-output transformation as a **pipeline of multiple neural network layers/blocks**.
  - If each component is differentiable (i.e. we can compute gradients), the whole model can be trained in an **end-to-end fashion**.
  - **End-to-end differentiable models:** all parameters of the entire model are jointly optimized to minimize a common loss(es).
    - One of the key aspects (advantage/power) of deep learning.
    - E.g. replaced human engineered, hand-crafted input feature engineering. These features were typically not optimized together with the model parameters. Now, it's part of the model.

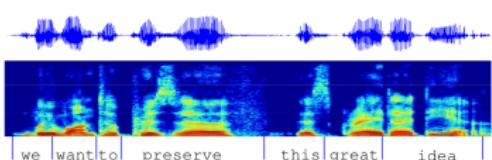
# Sequence to sequence problems

Many tasks are of type **sequence-to-sequence**:

An input sequence has to be transformed into an output sequence.

- Speech recognition: audio speech signals to written texts.
- Machine translation: source texts to target texts.
- Some mathematical problem solving (Assignment 4).

Automatic Speech Recognition



Handwriting Recognition



Machine Translation



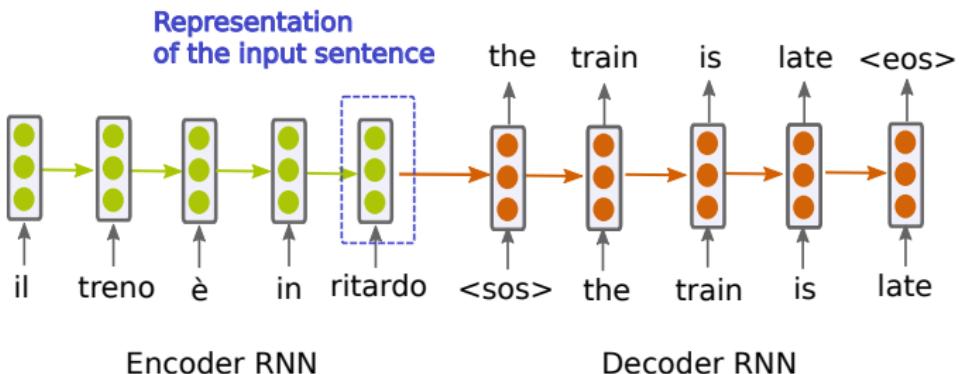
We will be using machine translation as a generic example to illustrate the problem.

Figures taken from [Ney 19].

# Encoder-Decoder Models

A generic model to solve sequence-to-sequence problems, w/ 2 RNNs:

- The first RNN **encodes** the input/source sequence.
- The final encoder-RNN state can be used to initialize the initial state for the second RNN.
- The second RNN **decodes** the output/target sequence.



# Encoder-Decoder Models, Training

You should feel that training this model is straightforward!

- During training, the target sentences are available.
- The decoder RNN can be trained just as a language model (shifted input/target).
- Use the cross entropy loss, only from the target sentence

# Encoder-Decoder Models, Training

You should feel that training this model is straightforward!

- During training, the target sentences are available.
- The decoder RNN can be trained just as a language model (shifted input/target).
- Use the cross entropy loss, only from the target sentence

What about “decoding”?

- At test time, you only have a “source” sentence available.
- ... and we want to extract the most likely target sentence according to the model.
- This is similar to what you are doing in Assignment 3 with language models.
- ... some extensions to improve decoding is possible.

# Encoder-Decoder Models, Beam search

- We want to find the most likely output sequence according to the model.
- Ideally, we should do exhaustive search: evaluate any possible sentences, the sentence with the highest probability is the answer.
- This is too expensive/impossible!
- Practical solutions:
  - **Greedy search**: take the argmax at each step (like in assignment 3).
  - better: **beam search** also known as top-K search.

# Encoder-Decoder Models, Beam search, illustration

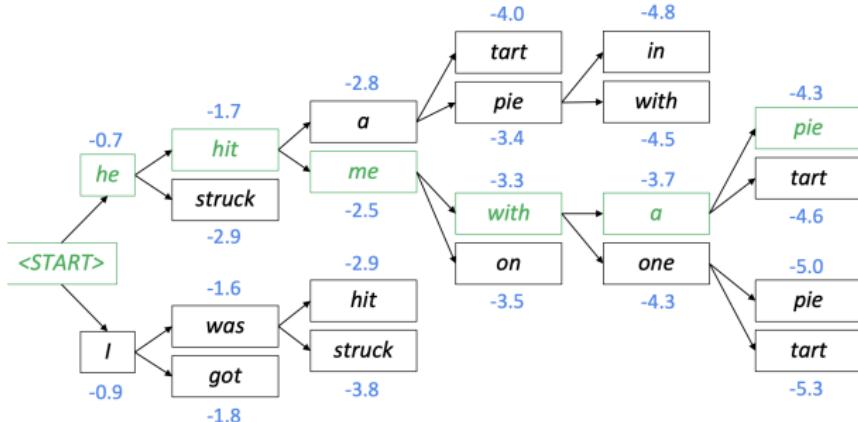


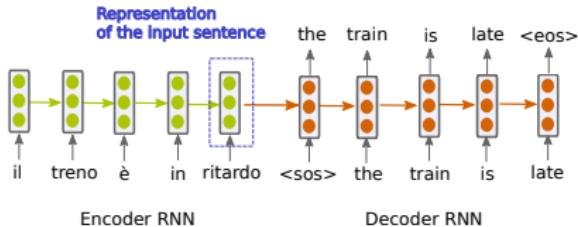
Figure taken from [Manning & See 19]. Top-2 decoding w/ beam size 2

- Keep expanding top  $K > 1$  hypotheses.
- Various end conditions are possible
  - e.g. length, number of hypotheses which reached sentence end token...
- By-product: N-best list, a list of top-N hypotheses.
  - You can check what other top hypotheses are! With that you can do e.g. model analyses, keyword detection over N-best list.

# Encoder-Decoder-Attention Models

Back to modeling...

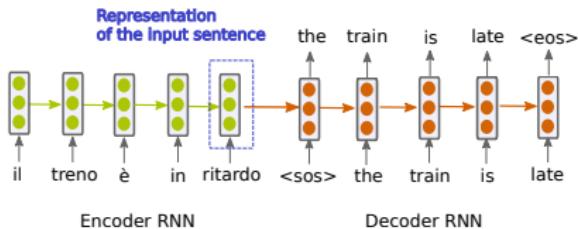
- RNNs are powerful general computers!
- Encoder-Decoder should just work!
- (maybe separation to encoder and decoder not even needed?)
- But practical limitation: hidden state size of RNN (memory size).



# Encoder-Decoder-Attention Models

Back to modeling...

- RNNs are powerful general computers!  
Encoder-Decoder should just work!  
(maybe separation to encoder and decoder not even needed?)
- But practical limitation: hidden state size of RNN (memory size).

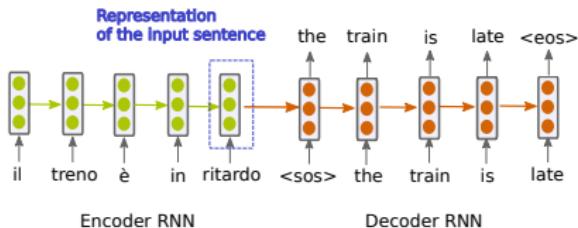


- **Attention** (Sec. 3.3) to rescue: augment decoder RNN with attention over encoder states.

# Encoder-Decoder-Attention Models

Back to modeling...

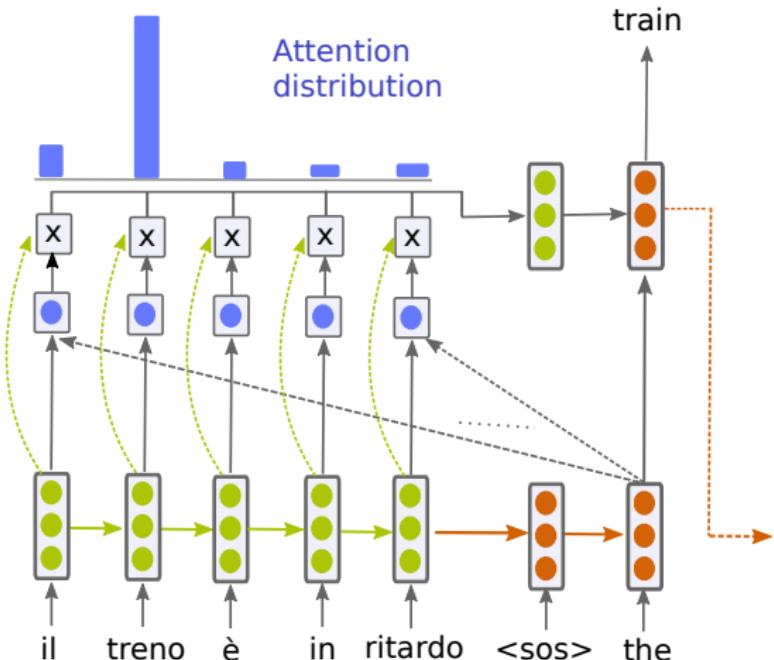
- RNNs are powerful general computers!  
Encoder-Decoder should just work!  
(maybe separation to encoder and decoder not even needed?)
- But practical limitation: hidden state size of RNN (memory size).



- **Attention** (Sec. 3.3) to rescue: augment decoder RNN with attention over encoder states.
  - ... process the input "piece-wise".
- Reminder: Attention is all about key/value/query manipulation. Here queries come from the decoder, and keys/values from the encoder.

# Encoder-Decoder-Attention Models, illustration

The model “focuses” on some parts of the input (encoder states) at every decoding step.



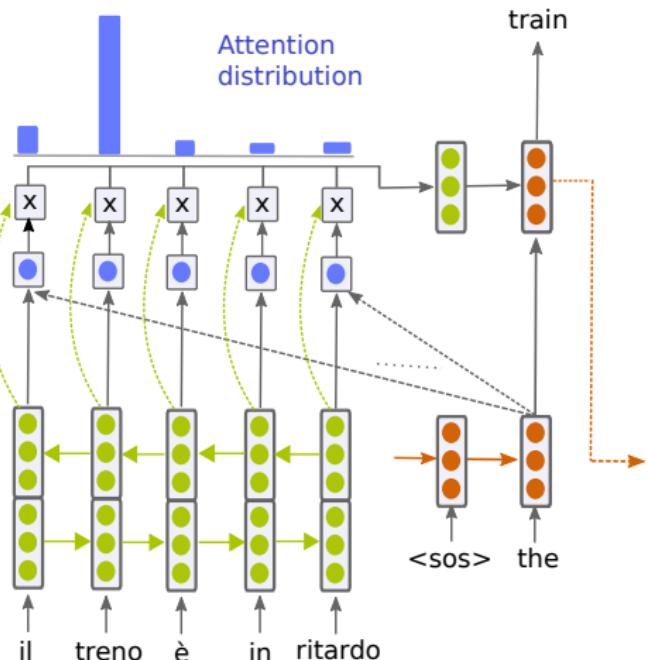
# Modification to Encoder, Bi-directionality

- One detail is “sub-optimal” in the previous figure.
- If a **standard/uni-directional RNN** is used for the encoder, the amount of information encoded in vectors at each encoder position is not equal.
  - Attention would very likely simply always focus on the vector containing the largest amounts of information.
  - Encoder vector at each position should contain contexts for both past and future words.

# Modification to Encoder, Bi-directionality

- One detail is “sub-optimal” in the previous figure.
- If a **standard/uni-directional** RNN is used for the encoder, the amount of information encoded in vectors at each encoder position is not equal.
  - Attention would very likely simply always focus on the vector containing the largest amounts of information.
  - Encoder vector at each position should contain contexts for both past and future words.
- Instead: we can use **bi-directional RNNs** in the encoder.
  - Essentially 2 RNNs, reading the inputs in forward and backward directions.
  - State vectors from forward and backward RNNs are **concatenated** to form the context vector for each encoder position.
  - In fact, this option is part of `torch.nn.LSTM` in PyTorch  
`bidirectional=True`.

# Modification to Encoder, Bi-directionality, illustration



# Encoder-Decoder-Attention Models, equations

NB: there are many variants with small differences. Here, one basic variant.  
Given sequences of:

- $S$  source tokens  $(x_1, \dots, x_S)$  with a source vocabulary  $x_s \in \mathbb{S}$
- $T$  target tokens  $(y_1, \dots, y_T)$  with a target vocabulary  $y_t \in \mathbb{T}$

Our model computes  $p(y_t | y_1^{t-1}, x_1^S)$

# Encoder-Decoder-Attention Models, equations

NB: there are many variants with small differences. Here, one basic variant.  
 Given sequences of:

- $S$  source tokens  $(x_1, \dots, x_S)$  with a source vocabulary  $x_s \in \mathbb{S}$
- $T$  target tokens  $(y_1, \dots, y_T)$  with a target vocabulary  $y_t \in \mathbb{T}$

Our model computes  $p(y_t | y_1^{t-1}, x_1^S)$

- The encoder transforms  $(x_1, \dots, x_S)$  to a **sequence of encoder state vectors**  $(h_1^{(\text{enc})}, \dots, h_S^{(\text{enc})})$ .
- Each  $h_s^{(\text{enc})} = [h_s^{(\text{fwd})}, h_s^{(\text{bwd})}]_{\text{feat}} \in \mathbb{R}^{2 \times d_{\text{enc}}}$  is a concatenation of two vectors along the feature dimension.  $d_{\text{enc}}$  is the encoder RNN state dimension.

# Encoder-Decoder-Attention Models, equations

NB: there are many variants with small differences. Here, one basic variant.  
 Given sequences of:

- $S$  source tokens  $(x_1, \dots, x_S)$  with a source vocabulary  $x_s \in \mathbb{S}$
- $T$  target tokens  $(y_1, \dots, y_T)$  with a target vocabulary  $y_t \in \mathbb{T}$

Our model computes  $p(y_t | y_1^{t-1}, x_1^S)$

- The encoder transforms  $(x_1, \dots, x_S)$  to a **sequence of encoder state vectors**  $(h_1^{(\text{enc})}, \dots, h_S^{(\text{enc})})$ .
- Each  $h_s^{(\text{enc})} = [h_s^{(\text{fwd})}, h_s^{(\text{bwd})}]_{\text{feat}} \in \mathbb{R}^{2 \times d_{\text{enc}}}$  is a concatenation of two vectors along the feature dimension.  $d_{\text{enc}}$  is the encoder RNN state dimension.
- $h_s^{(\text{fwd})}$  and  $h_s^{(\text{bwd})}$  are computed using two separate RNNs; with two different directions, forward (fwd) and backward (bwd).
  - i.e.  $[h_1^{(\text{fwd})}, \dots, h_S^{(\text{fwd})}] = \text{RNN}_{\text{fwd}}(x_1^S)$  and  $[h_S^{(\text{bwd})}, \dots, h_1^{(\text{bwd})}] = \text{RNN}_{\text{bwd}}(x_S^1)$

# Encoder-Decoder-Attention Models, equations

NB: there are many variants with small differences. Here, one basic variant.  
 Given sequences of:

- $S$  source tokens  $(x_1, \dots, x_S)$  with a source vocabulary  $x_s \in \mathbb{S}$
- $T$  target tokens  $(y_1, \dots, y_T)$  with a target vocabulary  $y_t \in \mathbb{T}$

Our model computes  $p(y_t | y_1^{t-1}, x_1^S)$

- The encoder transforms  $(x_1, \dots, x_S)$  to a **sequence of encoder state vectors**  $(h_1^{(\text{enc})}, \dots, h_S^{(\text{enc})})$ .
- Each  $h_s^{(\text{enc})} = [h_s^{(\text{fwd})}, h_s^{(\text{bwd})}]_{\text{feat}} \in \mathbb{R}^{2 \times d_{\text{enc}}}$  is a concatenation of two vectors along the feature dimension.  $d_{\text{enc}}$  is the encoder RNN state dimension.
- $h_s^{(\text{fwd})}$  and  $h_s^{(\text{bwd})}$  are computed using two separate RNNs; with two different directions, forward (fwd) and backward (bwd).
  - i.e.  $[h_1^{(\text{fwd})}, \dots, h_S^{(\text{fwd})}] = \text{RNN}_{\text{fwd}}(x_1^S)$  and  $[h_S^{(\text{bwd})}, \dots, h_1^{(\text{bwd})}] = \text{RNN}_{\text{bwd}}(x_S^1)$

Let's denote:  $H^{(\text{enc})} = [h_1^{(\text{enc})}, \dots, h_S^{(\text{enc})}]_{\text{time}} = \text{Encoder}(x_1, \dots, x_S)$

# Encoder-Decoder-Attention Models, equations (cont'd)

On the decoder side, we have the target tokens  $(y_1, \dots, y_T)$ .

- **Decoder RNN state**  $h_t^{(\text{dec})}$  is computed as:

$$h_t^{(\text{dec})} = \text{RNNCell}(h_{t-1}^{(\text{dec})}, y_{t-1}, c_t)$$

Just like the usual RNN but with an extra dependency on  $c_t$ .

# Encoder-Decoder-Attention Models, equations (cont'd)

On the decoder side, we have the target tokens  $(y_1, \dots, y_T)$ .

- **Decoder RNN state**  $h_t^{(\text{dec})}$  is computed as:

$$h_t^{(\text{dec})} = \text{RNNCell}(h_{t-1}^{(\text{dec})}, y_{t-1}, c_t)$$

Just like the usual RNN but with an extra dependency on  $c_t$ .

- The **context vector**  $c_t$  is computed by attention (Sec. 3.3) where you use  $h_{t-1}^{(\text{dec})}$  as **query**, and  $H^{(\text{enc})}$  as **key** and **value** vectors.

$$c_t = \text{Attention}(H^{(\text{enc})}, H^{(\text{enc})}, h_{t-1}^{(\text{dec})})$$

- The current decoder state  $h_t^{(\text{dec})}$  is fed to the softmax layer to get the output  $p(y_t | y_1^{t-1}, x_1^S)$ .

# Encoder-Decoder-Attention Models, equations (cont'd)

On the decoder side, we have the target tokens  $(y_1, \dots, y_T)$ .

- **Decoder RNN state**  $h_t^{(\text{dec})}$  is computed as:

$$h_t^{(\text{dec})} = \text{RNNCell}(h_{t-1}^{(\text{dec})}, y_{t-1}, c_t)$$

Just like the usual RNN but with an extra dependency on  $c_t$ .

- The **context vector**  $c_t$  is computed by attention (Sec. 3.3) where you use  $h_{t-1}^{(\text{dec})}$  as **query**, and  $H^{(\text{enc})}$  as **key** and **value** vectors.

$$c_t = \text{Attention}(H^{(\text{enc})}, H^{(\text{enc})}, h_{t-1}^{(\text{dec})})$$

- The current decoder state  $h_t^{(\text{dec})}$  is fed to the softmax layer to get the output  $p(y_t | y_1^{t-1}, x_1^S)$ .

If interested, more reading: [Wu & Schuster<sup>+</sup> 16] *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*

# Transformer models

Today's state-of-the-art model for translation.

- "Replace" RNN layers by Transformer layers.

*Attention is all you need...*

# Transformer models

Today's state-of-the-art model for translation.

- "Replace" RNN layers by Transformer layers.  
*Attention is all you need...*
- The positional encoding is needed to both encoder and decoder inputs.
- The decoder layers accesses the encoder states using attention.

# Transformer models

Today's state-of-the-art model for translation.

- "Replace" RNN layers by Transformer layers.  
*Attention is all you need...*
- The positional encoding is needed to both encoder and decoder inputs.
- The decoder layers access the encoder states using attention.
- Compared to what we saw in **Sec 3.3** (decoder-only/auto-regressive Transformer), this would require an **extra attention layer**. Each Transformer decoder layer consists of:
  - One self-attention layer (key/value from **previous target tokens**)
  - One attention layer (key/value from **encoder states**)
  - One feed-forward layer

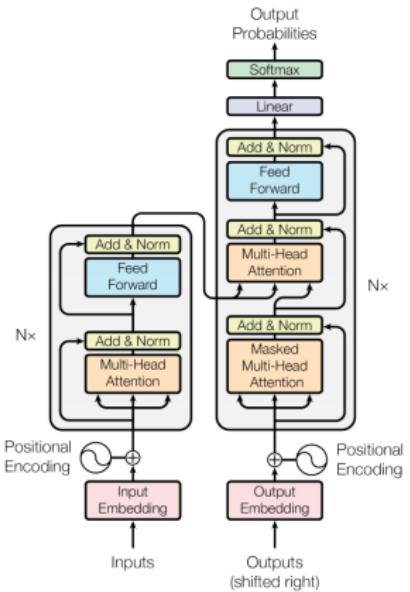


Figure from [Vaswani & Shazeer<sup>+</sup> 17].

# Transformer models (cont'd)

- All core computations are based on attention.  
Reminder:  $\text{Attention}(K, V, Q) = V \text{ softmax}_{\text{time}}(K^T Q)$
- The difference between the **3 types** of attention in the Transformer are the choice of keys  $K$ , values  $V$ , and queries  $Q$  used in the computation.
  - **Self-attention in the encoder:**  
the whole input sequence produces all key, value and query vectors.
  - **Decoder-to-encoder attention in the decoder:**  
queries are decoder states, key and values are from the encoder states.
  - **Self-attention in the decoder:**  
query is computed from the current input to the decoder, keys and values are the previous decoder states (just like in language models)

# Batch and masking

## ■ Batch

- Sequences with different lengths can be in the same batch.
  - We need to do padding (just like before) for **both source and target sequences**.
  - Reminder: padded parts (here on the target side) should be excluded (masked) from the computation of the loss.
- For all attention types, the unnormalized attention weights  $K^T Q$  can be computed in a single matrix multiplication (including all positions).
- Some positions are not valid: positions that the attention are not supposed to access can be **masked** afterwards: ie. attention scores of the masked positions are set to zero.

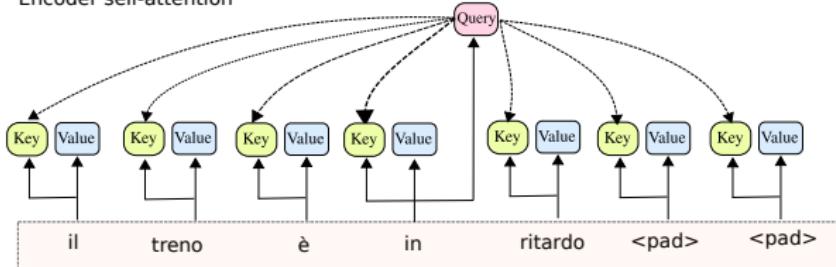
## ■ Masks for attention

- For each attention function, we need to specify a mask.
- The attention scores are computed for all positions in the batch.
- The parts which should not be attended should be masked, i.e. the attention score should be set to zero (or -infinity before softmax).

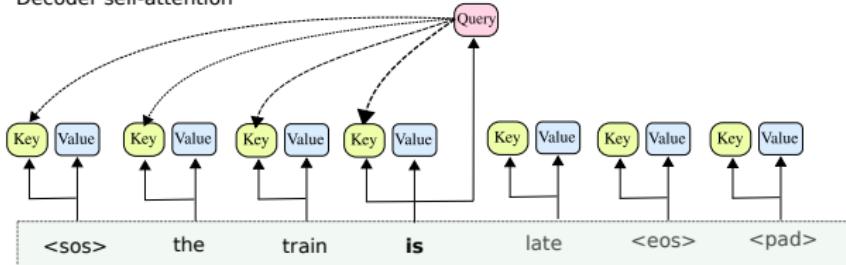
# Attention types, Illustrations

Again with an example for Italian to English translation.

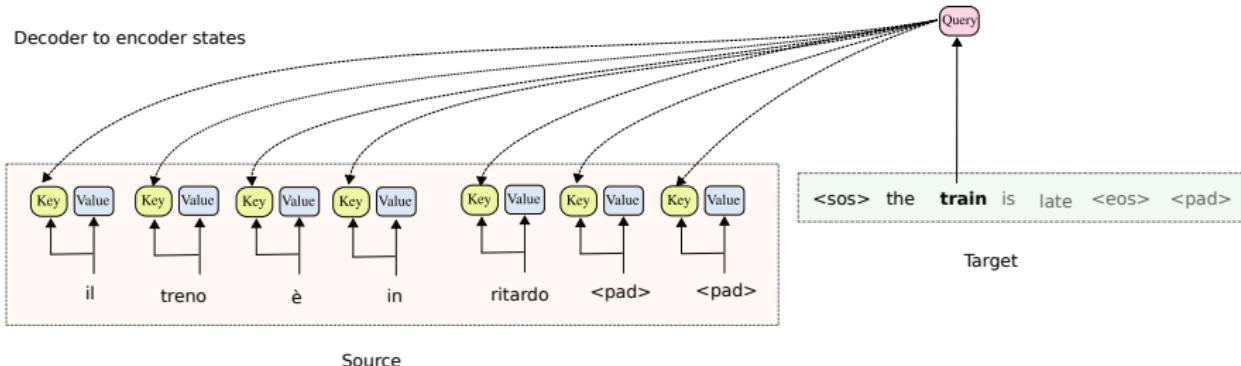
Encoder self-attention



Decoder self-attention



# Attention, Illustrations (cont'd)



NB:

- <sos> and <eos> are crucial for the sequence on the **target** side (we could also have them for the source side sequence; not illustrated here).

## A few words on nn.Transformer

- The official one from PyTorch, which you will use in the final assignment!
  - Maybe not the most user friendly implementation...
  - Make sure to use the latest PyTorch version! ( $\geq 1.6.0$ )
- nn.Transformer implements the full **encoder-decoder model**.
  - It internally calls TransformerEncoderLayer, etc...
  - Full documentation at  
<https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html>
  - Code: [https://pytorch.org/docs/stable/\\_modules/torch/nn/modules/transformer.html](https://pytorch.org/docs/stable/_modules/torch/nn/modules/transformer.html)
  - The constructor is standard: you specify Transformer's hyper-parameters.
- NB: remember we've already learned **positional encoding** in Sec. 3.3 with an example implementation. **Do not forget it!**
- There are a couple of things you have to think, to use the forward function.

## nn.Transformer, forward (cont'd)

There are **6 mask tensors** you can feed to the `forward` function.

■ **3 masks for padding.** You need them all (in principle):

- `src_key_padding_mask`
- `tgt_key_padding_mask`
- `memory_key_padding_mask`

**But** `memory` refers to encoder states for decoder-to-encoder attention. So the same mask can be used for `src_key_padding_mask` and `memory_key_padding_mask`.

■ **3 attention masks** for limiting the attention range:

- `src_mask`
- `tgt_mask`
- `memory_mask`

**But** only `tgt_mask` has to be specified (for this assignment).

■ Method to generate `tgt_mask` is implemented in `torch.nn.Transformer`: see `generate_square_subsequent_mask`.

■ You have a choice of create masks with values: `-inf/0` or `True/False` to specify mask/no-mask respectively (if you use the latest PyTorch version).

**Be careful with the meaning of True/False vs. mask/no-mask.**

## nn.Transformer, forward (cont'd)

- tgt argument of forward function expects the **input sequence to the decoder**.
- Reminder, it is like in language modeling:
  - If the target side sequence is: <sos> a b <eos>
  - The input to the decoder should be the sequence: <sos> a b
  - While the expected output sequence (to compute the loss) is: a b <eos>

Make sure to correctly do the shifting!

# Hints for Assignment 4, Summary

- Good news: Model itself is implemented in the helper code!
- Be careful with sequences (shift) to be used in the tgt argument of forward function and the loss computation.
- Be careful with all the **masks** for forward function.
- Method to generate tgt\_mask is implemented in `torch.nn.Transformer: generate_square_subsequent_mask`.  
You can test it separately.
- In general, be careful with the meaning of the mask values (e.g. does 0 means mask or no-mask?)
- To implement **greedy search**:
  - Check how the encoder and decoder submodules are called in `forward`.  
`https://pytorch.org/docs/stable/_modules/torch/nn/modules/transformer.html`
  - Remember that you can access the encoder and decoder via `.encoder` and `.decoder` once you have your Transformer object.

# Important announcement for Assignment 4

■ The deadline is **January 15, 2023** (Sunday at 10 pm).

## ■ Why not 2(3)-week deadline as usual?

- To give you more time (possibility to work during the holidays if you wish).
- This does not mean that you need more than 2 weeks to solve the problem. You are welcome to finish within the usual 2-week period and submit before Christmas.

## ■ Warning:

- We know that this date is in the middle of the exam period.
- But should be better than December 23.
- There will be no extension to the deadline.
- The late submission policy will apply as usual.
- We will answer questions via email until the end of December, but very likely not at all in January: Be aware of this, and ask your questions early!

# Important announcement for Assignment 4 (cont'd)

- We will have QA sessions for A4:
  - Dec 12 (second part)
  - Dec 19 (full session).
- In addition, you should not hesitate to contact us by email (even with your code).
- Reminder:
  - Please **always** put all TAs in CC, in any email.
  - Please do not send messages via MS Chat (except during the class).

## What have we learned?

- Building models
- Examples of end-to-end differentiable models.
- Encoders, decoders
- Sequence-to-sequence problems and beyond.

## Coming up next...

- More on practical aspects for training (Chapter 5).
- ...

# Outline

1. Workflow Overview
2. Introduction to Tools
3. Fundamental building blocks
4. Building models
- 5. Practical tricks and methods for training**
6. Final words and Outlook

# Practical tricks?

- There are many practical aspects to be correctly configured to obtain good neural network based models.
- You must have already seen many of these concepts in the exercises & assignments.

# Model parameter initialization

- General principle: initialize with small, random numbers.

```
torch.nn.init.normal_(x, mean=0.0, std=0.01),  
torch.nn.init.uniform_(x, a=-0.1, b=0.1)
```

- More sophisticated methods exist (dependence on input/output dimensions), see e.g.

  - Xavier initializer `torch.nn.init.xavier_uniform_`
  - Variance scaling initializer `torch.nn.init.kaiming_uniform_`

- In general: do not overthink, use some standard setups.

- Except when you want to specify how some model components behave at the beginning of training:

  - e.g. bias initialization for LSTM's forget gate.
  - or when addressing some specific problems...

# Model parameter initialization, example

```
1 def init_weights(m):
2     if type(m) == nn.Linear:
3         torch.nn.init.xavier_uniform_(m.weight)
4         m.bias.data.fill_(0.01)
5
6 net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
7 net.apply(init_weights)
```

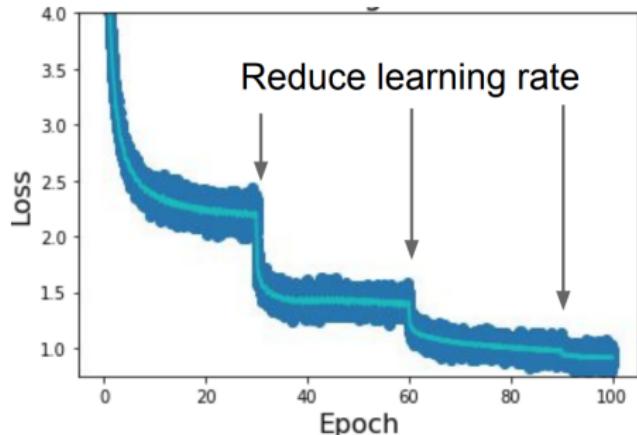
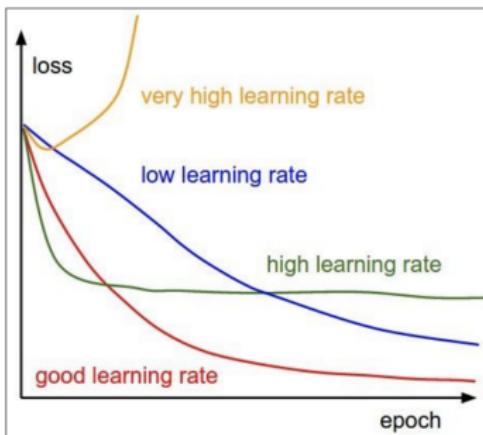
- Note: `nn.Sequential` is another way to create a model.
- `.apply` applies the function to all sub-modules.
- Or do it inside `__init__` function of your model.

Have you ever checked how it is done by default in e.g. `nn.Linear` ?

[https://pytorch.org/docs/stable/\\_modules/torch/nn/modules/linear.html](https://pytorch.org/docs/stable/_modules/torch/nn/modules/linear.html)

# Choice of learning rate

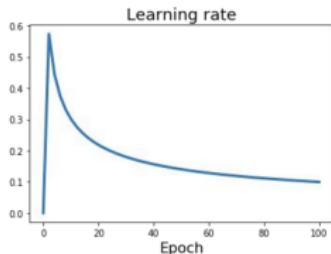
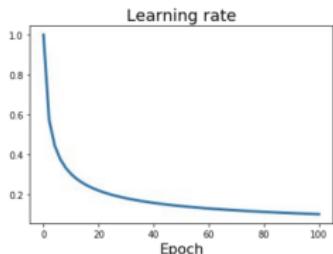
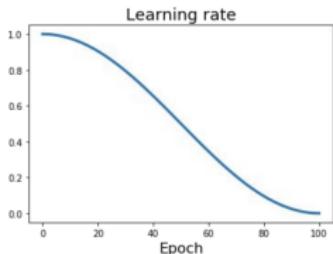
- Learning rate is typically the most important training hyper-parameter.
- Different learning rates, different training behaviors.
- Good strategy: modify learning rate during training.



Figures taken from [Li & Johnson<sup>+</sup> 19b].

# Learning rate scheduling

## How to modify the learning rate during training?



Figures taken from [Li & Johnson<sup>+</sup> 19b].

- Or better: change dependent on performance on the **validation set**.
- If improvement on the validation set is less than  $x\%$ , then reduce the learning rate by some factor.

# Learning rate scheduling, example

- `torch.optim.lr_scheduler` provides different strategies.  
`ReduceLROnPlateau`, `CosineAnnealingLR`, ...

```
1 optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
2 scheduler =
3     torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min')
4
5 for epoch in range(10):
6     train(...)
7     val_loss = validate(...)
8     # Note that step should be called after validate()
9     scheduler.step(val_loss)
```

# Depending on optimizers

- The range of good choice of initial learning rate depends on the optimizer.
- Common optimizers: SGD and Adam.
- Typical configurations/heuristics:
  - SGD: rather high learning rate (e.g. 1, 0.1) typically with gradient clipping (next page).
  - Adam: rather small learning rate (e.g. 1e-3, 1e-4).

Also Adam often requires none or only some minimum learning rate scheduling.

# Other training helper tricks

Techniques to **stabilize training**, allowing us to use **high learning rates** without getting the model to crash (NaN loss value):

## ■ Gradient clipping

- There are a couple of ways to do it.
- Common approach: global norm clipping.
  - One hyper-parameter: max gradient norm  $M$ .
  - If norm computed over all gradients is larger than  $M$ , scale all gradients by  $M/\text{norm}$ .

```
1 clip_val = 1.0
2
3 loss.backward()
4 torch.nn.utils.clip_grad_norm_(model.parameters(), clip_val)
5 optimizer.step()
```

# Other training helper tricks (cont'd)

- **Normalization layers:** transform input  $x$  to  $y$  via:

$$y = \frac{x - \text{E}[x]}{\sqrt{\text{Var}[x]}} * a + b$$

where  $a$  and  $b$  are vectors and learnable parameters.

Different methods depending on the support of computing mean  $\text{E}[x]$  and variance  $\text{Var}[x]$ :

- **Layer normalization:**

→ mean and variance computed along the **feature dimension**.

- **Batch normalization**

→ mean and variance computed per-dimension over the **mini-batches**.

→ during training: keeps updating estimates of its computed mean and variance, which are then used during evaluation.

# Choosing the model size

- **Possible strategy:** put as many trainable parameters as possible, until the model overfits.
- Apply **regularization techniques** and make it even larger.
- Also: larger dataset (or data augmentation) → larger model size.

# Overfitting & Early stopping

- (For most tasks) When your setup is correct, and the model has **enough capacity** (large enough number of parameters), your model should get very good performance on the **training data** when trained for long enough.
- Typically, performance on validation set degrades at some point (again, if model is large enough!): model **overfits** to training data.
- Good strategy: **stop training earlier, by checking validation error.**

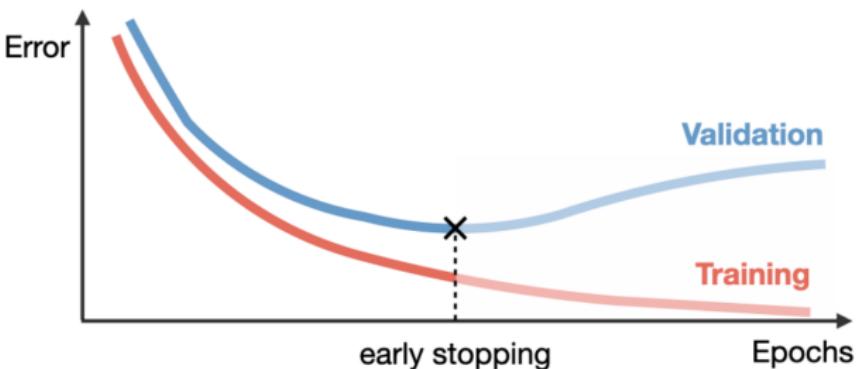


Figure taken from [Amidi & Amidi 19].

- Standard machine learning regularization techniques can be applied to neural networks.
  - E.g., extra loss on the weights: minimize its L1, L2 norms...
  - In PyTorch, all standard optimizer has an option for *weight decay*:

```
1 optimizer = torch.optim.SGD(  
2     model.params(), lr=0.01, weight_decay=1e-4)
```

- which is equivalent to L2 regularization for many optimizers (not for Adam).
- Extra training hyper-parameter...

# Dropout

- Popular regularization methods for neural networks: **dropout**.

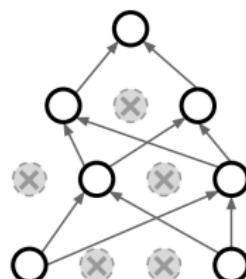
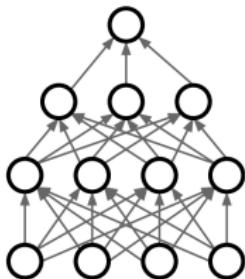


Figure taken from [Li & Johnson<sup>+</sup> 19b].

- One hyper-parameter: dropout rate  $p$ .
- Set each activation to 0 with a probability of  $p$  during training.
- At test time: use all activations but scale by  $1 - p$ .
  - (Or scale by  $1/(1 - p)$  during training instead, and do nothing at test time).
- Note: introduce different model behaviors in train/eval modes. PyTorch reminder: `model.train()` and `model.eval()` .

# Dropout, implementations

- Dropout can be defined just as a regular layer (we learnt this in A2):

```
1 >>> input = torch.randn(2, 6)
2 >>> m = nn.Dropout(p=0.5)
3 >>> output = m(input)
4 >>> input
5 tensor([[-0.0863,  0.6893,  0.0138, -2.0447,  1.1345,  0.3555],
6          [ 1.1229,  0.0027,  0.0993, -0.7476,  0.5680, -0.4255]])
7 >>> output
8 tensor([[-0.1726,  1.3785,  0.0276, -0.0000,  2.2689,  0.7110],
9          [ 0.0000,  0.0054,  0.1985, -1.4953,  1.1359, -0.0000]])
```

- Note: some Pytorch build-in layers/modules already have an internal dropout option.

# Model averaging, for evaluation

- **Ensembling:** combine multiple models' output distributions for **evaluation**.  
e.g. combination by weighted averaging.  
Model combinations should never hurt!
- **Model averaging:** average weights of different models with the same architecture.  
In some cases helpful to average different model checkpoints in the same training run (relation to some special optimization algorithm...).

# Outline

1. Workflow Overview
2. Introduction to Tools
3. Fundamental building blocks
4. Building models
5. Practical tricks and methods for training
- 6. Final words and Outlook**

# Summary

What have we learned?

# Looking back

In assignments and exercises, you have implemented:

- Image classification with convolutional neural networks.
- Language modeling with RNNs.
  - as an example of sequence processing/generation problems.
- Mathematical problem solving with Transformers
  - as an example of sequence to sequence problems.

These are real hands-on experiences.

## Looking back (cont'd)

You put hands on practical problems and acquired experience:

- PyTorch
- Sense of system components: data, model, loss, optimization...
- Hyper-parameter tuning: Number of layers, hidden layer size...
- Training batch construction: batch size, padding, ...
- ...

## Looking back (cont'd)

You put hands on practical problems and acquired experience:

- PyTorch
- Sense of system components: data, model, loss, optimization...
- Hyper-parameter tuning: Number of layers, hidden layer size...
- Training batch construction: batch size, padding, ...
- ...

Building blocks you learned are used in many popular/real/commercial applications, e.g.:

- Machine translation
- OpenAI's GPT-3 language model.
- ...

# Generic approach for any Seq2Seq tasks

- You've learned seq2seq processing
- Many tasks are "translation" like.
- + some task specific/engineering tweaks.
  - E.g. speech recognition: input sequence (audio) is very long.
  - Downsampling is done using convolution or pooling in the encoder in order to reduce the effective sequence length for RNNs/Transformers.

## Other Seq2Seq tasks, e.g., with images

**Principle of encoding** a source information into a vector, and pass it to a decoder/classifier is general. Examples:

- Image captioning
  - Input: image → encode it with convnets.
  - Output: text describing image → generate it with RNNs or Transformers.
- Image question answering
  - Inputs: image and text/question.
  - Output: text or just answer token.

You can design some task specific models for the best performance, but the idea of encoding/decoding is general.

## More topics...

- We can not cover all interesting works/models!
- This final chapter: small catalog so that you hear some keywords!
- The rest is up to your own curiosity and interests.
- Search and read more about what you are interested in!

# Reinforcement learning

A big chapter uncovered in this lecture.

- Particularly impressive performance for game playing:
- AlphaGo [Silver & Huang<sup>+</sup> 16], StarCraft II [Vinyals & Babuschkin<sup>+</sup> 19],...
- Robotics: object manipulation [Andrychowicz & Baker<sup>+</sup> 20]...
- Also many practical aspects...

Another big chapter not covered in this lecture.

- Only a model for generating texts has been presented (assignment 3)
- Models which *generate* something: image, text, speech, music, ..

You can search for:

- Generative adversarial networks (GANs) [Goodfellow & Pouget-Abadie<sup>+</sup> 14]
- Variational auto-encoders (VAEs) [Kingma & Welling 14]
- Flow based models [Dinh & Krueger<sup>+</sup> 15, Rezende & Mohamed 15]
- Auto-regressive models, such as PixelRNN [van den Oord & Kalchbrenner<sup>+</sup> 16], WaveNet [Oord & Dieleman<sup>+</sup> 16], ...
- **Very hot in 2022:** diffusion models (see examples in the first class)

# Self-supervised learning

- Subset of unsupervised learning. Recently became very popular.
- Typically: take some data, mask parts of it, and ask the model to predict the hidden parts given the rest.
  - Originally proposed for texts/natural language processing.
  - Models are often based on Transformers.
  - Key models: BERT [Devlin & Chang<sup>+</sup> 19] and its variants.
  - Also the standard language models fall into this category.
- Now also applied beyond NLP: images, videos,...
- Also related: contrastive learning.

# Final words

- By now: You should know the basics to tackle many problems.
- If there is any topic/idea which interests you, search (public implementations) and read papers, and implement/modify it!

# Acknowledgements, Resources & References

Contents of this slides are partially adapted from:

- Paulo Rauber's materials for the DLL (2018, 2019).  
[http://paulorrauber.com/slides/deep\\_learning\\_lab.pdf](http://paulorrauber.com/slides/deep_learning_lab.pdf)
- Princeton, NLP class PyTorch.
- Stefan Otte's tutorial "Practical PyTorch" (2017):  
[https://github.com/sotte/pytorch\\_tutorial](https://github.com/sotte/pytorch_tutorial)
- David Völgyes's lecture (2020):  
[https://www.uio.no/studier/emner/matnat/ifi/IN5400/v20/material/lectureslides/in5400\\_week4\\_2020\\_pytorch\\_lecture4.pdf](https://www.uio.no/studier/emner/matnat/ifi/IN5400/v20/material/lectureslides/in5400_week4_2020_pytorch_lecture4.pdf)
- Lecture note from Fei-Fei Li's team (Stanford):  
<http://cs231n.stanford.edu/slides/2019/>

# Acknowledgements, Resources & References

[Amidi & Amidi 19] A. Amidi, S. Amidi.

Cheatsheets, stanford cs230: Deep learning.

[https:](https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks)

//stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks,  
Winter, 2019.

[Andrychowicz & Baker<sup>+</sup> 20] O. M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray et al.

Learning dexterous in-hand manipulation.

*The International Journal of Robotics Research*, Vol. 39, No. 1, pp. 3–20, 2020.

[Bahdanau & Cho<sup>+</sup> 15] D. Bahdanau, K. Cho, Y. Bengio.

Neural machine translation by jointly learning to align and translate.

In *Int. Conf. on Learning Representations (ICLR)*, San Diego, CA, USA, May 2015.

[Devlin & Chang<sup>+</sup> 19] J. Devlin, M. Chang, K. Lee, K. Toutanova.

BERT: pre-training of deep bidirectional transformers for language understanding.

In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pp. 4171–4186, Minneapolis, MN, USA, June 2019.

[Dieleman 20] S. Dieleman.

Deep learning lectures DeepMind-UCL, part 3.

<https://www.youtube.com/watch?v=shVKhOmT0HE>, 2020.

[Dinh & Krueger<sup>+</sup> 15] L. Dinh, D. Krueger, Y. Bengio.

NICE: non-linear independent components estimation.

# Acknowledgements, Resources & References

In *International Conference on Learning Representations (ICLR), Workshop Track Proceedings*, San Diego, CA, USA, May 2015.

[Finn & Yu<sup>+</sup> 17] C. Finn, T. Yu, T. Zhang, P. Abbeel, S. Levine.

One-shot visual imitation learning via meta-learning.

In *Conference on Robot Learning (CoRL)*, Proc. Machine Learning Research, pp. 357–368, Mountain View, CA, USA, Nov. 2017.

[Gehring & Auli<sup>+</sup> 17] J. Gehring, M. Auli, D. Grangier, D. Yarats, Y. N. Dauphin.

Convolutional sequence to sequence learning.

In *Proc. Int. Conf. on Machine Learning (ICML)*, pp. 1243–1252, Sydney, Australia, Aug. 2017.

[Gers & Schmidhuber<sup>+</sup> 00] F. A. Gers, J. Schmidhuber, F. Cummins.

Learning to forget: Continual prediction with LSTM.

*Neural computation*, Vol. 12, No. 10, pp. 2451–2471, 2000.

[Goodfellow & Pouget-Abadie<sup>+</sup> 14] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio.

Generative adversarial nets.

In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pp. 2672–2680. Montréal, Canada, Dec. 2014.

[He & Zhang<sup>+</sup> 16a] K. He, X. Zhang, S. Ren, J. Sun.

Deep residual learning for image recognition.

In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, Las Vegas, NV, USA, June 2016.

# Acknowledgements, Resources & References

- [He & Zhang<sup>+</sup> 16b] K. He, X. Zhang, S. Ren, J. Sun.  
Identity mappings in deep residual networks.  
In *Proc. European Conf. on Computer Vision (ECCV)*, pp. 630–645, Amsterdam, Netherlands, Oct. 2016.
- [He & Zhang<sup>+</sup> 16c] K. He, X. Zhang, S. Ren, J. Sun.  
Slides, deep residual learning for image recognition.  
[http://kaiminghe.com/cvpr16resnet/cvpr2016\\_deep\\_residual\\_learning\\_kaiminghe.pdf](http://kaiminghe.com/cvpr16resnet/cvpr2016_deep_residual_learning_kaiminghe.pdf), 2016.
- [Hochreiter & Schmidhuber 97] S. Hochreiter, J. Schmidhuber.  
Long short-term memory.  
*Neural computation*, Vol. 9, No. 8, pp. 1735–1780, 1997.
- [Huang & Liu<sup>+</sup> 17] G. Huang, Z. Liu, L. van der Maaten, K. Q. Weinberger.  
Densely connected convolutional networks.  
In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, Honolulu, HI, USA, July 2017.
- [Kim & Kang<sup>+</sup> 18] K. Kim, S. Kang, J. Yoo, Y. Kwon, Y. Nam, D. Lee, I. Kim, Y.-S. Choi, Y. Jung, S. Kim et al.  
Deep-learning-based inverse design model for intelligent discovery of organic molecules.  
*npj Computational Materials*, Vol. 4, No. 1, pp. 1–7, 2018.
- [Kim 14] Y. Kim.  
Convolutional neural networks for sentence classification.  
In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1746–1751, Doha, Qatar, Oct. 2014.
- [Kingma & Ba 15] D. P. Kingma, J. Ba.

# Acknowledgements, Resources & References

Adam: A method for stochastic optimization.

In *Proc. Int. Conf. on Learning Representations (ICLR)*, San Diego, CA, USA, May 2015.

[Kingma & Welling 14] D. P. Kingma, M. Welling.

Auto-encoding variational bayes.

In *Int. Conf. on Learning Representations (ICLR)*, Banff, Canada, April 2014.

[Krizhevsky & Sutskever<sup>+</sup> 12] A. Krizhevsky, I. Sutskever, G. E. Hinton.

Imagenet classification with deep convolutional neural networks.

In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pp. 1106–1114, Lake Tahoe, NV, USA, Dec. 2012.

[Lang & Witbrock 88] K. J. Lang, M. J. Witbrock.

Learning to tell two spirals apart.

In *Proc. Connectionist Models Summer School*, pp. 52–59, Carnegie Mellon University, June 1988.

[Le & Venkatesh 20] H. Le, S. Venkatesh.

Neurocoder: Learning general-purpose computation using stored neural programs.

Preprint arXiv:2009.11443, 2020.

[LeCun & Cortes<sup>+</sup> 98] Y. LeCun, C. Cortes, C. J. Burges.

The mnist database of handwritten digits.

URL <http://yann.lecun.com/exdb/mnist>, 1998.

[Li & Johnson<sup>+</sup> 17] F.-F. Li, J. Johnson, S. Yeung.

Stanford cs231n: Convolutional neural networks for visual recognition, lecture 11.

[http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture11.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture11.pdf), Spring, 2017.

# Acknowledgements, Resources & References

[Li & Johnson<sup>+</sup> 19a] F.-F. Li, J. Johnson, S. Yeung.

Stanford cs231n: Convolutional neural networks for visual recognition, lecture 5.

[http://cs231n.stanford.edu/slides/2019/cs231n\\_2019\\_lecture05.pdf](http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture05.pdf), Spring, 2019.

[Li & Johnson<sup>+</sup> 19b] F.-F. Li, J. Johnson, S. Yeung.

Stanford cs231n: Convolutional neural networks for visual recognition, lecture 8.

[http://cs231n.stanford.edu/slides/2019/cs231n\\_2019\\_lecture08.pdf](http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture08.pdf), Spring, 2019.

[Luong & Pham<sup>+</sup> 15] T. Luong, H. Pham, C. D. Manning.

Effective approaches to attention-based neural machine translation.

In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1412–1421, Lisbon, Portugal, Sept. 2015.

[Manning & See 19] C. Manning, A. See.

Stanford cs224n/ling284: Natural language processing with deep learning, lecture 8.

<https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/slides/cs224n-2019-lecture08-nmt.pdf>, Winter, 2019.

[Mnih & Kavukcuoglu<sup>+</sup> 13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller.

Playing atari with deep reinforcement learning.

NIPS Deep Learning Workshop, Dec. 2013.

[Ney 19] H. Ney.

Lecture slides, advanced topics in machine learning and human language technology.

RWTH Aachen, Germany, 2019.

# Acknowledgements, Resources & References

- [Nikolaus & Abdou<sup>+</sup> 19] M. Nikolaus, M. Abdou, M. Lamm, R. Aralikatte, D. Elliott.  
Compositional generalization in image captioning.  
In *Proc. Computational Natural Language Learning (CoNLL)*, pp. 87–98, Hong Kong, China, Nov. 2019.
- [Oord & Dieleman<sup>+</sup> 16] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, K. Kavukcuoglu.  
Wavenet: A generative model for raw audio.  
Preprint arXiv:1609.03499, 2016.
- [Papineni & Roukos<sup>+</sup> 02] K. Papineni, S. Roukos, T. Ward, W. Zhu.  
BLEU: a method for automatic evaluation of machine translation.  
In *Proc. Association for Computational Linguistics (ACL)*, pp. 311–318, Philadelphia, PA, USA, July 2002.
- [Rezende & Mohamed 15] D. J. Rezende, S. Mohamed.  
Variational inference with normalizing flows.  
In *Proc. Int. Conf. on Machine Learning (ICML)*, pp. 1530–1538, Lille, France, July 2015.
- [Silver & Huang<sup>+</sup> 16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot et al.  
Mastering the game of go with deep neural networks and tree search.  
*Nature*, Vol. 529, No. 7587, pp. 484–489, 2016.
- [Srivastava & Greff<sup>+</sup> 15] R. K. Srivastava, K. Greff, J. Schmidhuber.  
Training very deep networks.  
In *Advances in Neural Information Processing Systems 28*, pp. 2377–2385, Montréal, Canada, Dec. 2015.

# Acknowledgements, Resources & References

[Stevens & Antiga 20] E. Stevens, L. Antiga.

*Deep learning with PyTorch.*

Manning Publications, 2020.

[van den Oord & Kalchbrenner<sup>+</sup> 16] A. van den Oord, N. Kalchbrenner, K. Kavukcuoglu.

Pixel recurrent neural networks.

In *Proc. Int. Conf. on Machine Learning (ICML)*, Vol. 48, pp. 1747–1756, New York City, NY, USA, June 2016.

[Vaswani & Shazeer<sup>+</sup> 17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin.

Attention is all you need.

In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pp. 5998–6008, Long Beach, CA, USA, Dec. 2017.

[Vinyals & Babuschkin<sup>+</sup> 19] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev et al.

Grandmaster level in starcraft ii using multi-agent reinforcement learning.

*Nature*, Vol. 575, No. 7782, pp. 350–354, 2019.

[Waibel & Hanazawa<sup>+</sup> 89] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, K. J. Lang.

Phoneme recognition using time-delay neural networks.

*IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 37, No. 3, pp. 328–339, 1989.

[Wu & Schuster<sup>+</sup> 16] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey et al.

Google's neural machine translation system: Bridging the gap between human and machine translation.

# Acknowledgements, Resources & References

Preprint arXiv:1609.08144, 2016.

[Xie & Girshick<sup>+</sup> 17] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, K. He.

Aggregated residual transformations for deep neural networks.

In *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5987–5995, Honolulu, HI, USA, July 2017.

[Zagoruyko & Komodakis 16] S. Zagoruyko, N. Komodakis.

Wide residual networks.

In *Proc. British Machine Vision Conference (BMVC)*, York, UK, Sept. 2016.

[Zhu & Park<sup>+</sup> 17] J. Zhu, T. Park, P. Isola, A. A. Efros.

Unpaired image-to-image translation using cycle-consistent adversarial networks.

In *IEEE International Conference on Computer Vision (ICCV)*, pp. 2242–2251, Venice, Italy, Oct. 2017.

# End

Instructor: Kazuki Irie

TAs: Róbert Csordás, Aditya Ramesh

[kazuki.irie@usi.ch](mailto:kazuki.irie@usi.ch), [robert.csordas@idsia.ch](mailto:robert.csordas@idsia.ch), [aditya.ramesh@idsia.ch](mailto:aditya.ramesh@idsia.ch)