Assignment 1: Let the Tensors Flow

In this assignment, you will implement and train your first deep model on a well-known image classification task. You will also familiarize yourself with the Tensorflow (https://www.tensorflow.org/) library we will be using for this course.

General Assignment Notes

- You will need to do some extra reading for these assignments. Sorry, but there is no way around this. This should be significantly reduced by attending the practical exercise sessions, but things are re-iterated here for people who missed them (that's why there's so much text ;)).
- Assignments are posed in a very open-ended manner. Often you only "need" to complete a rather basic task. However, you will get *far* more out of this class by going beyond these basics. Some suggestions for further explorations are usually contained in the assignment description. Ideally though, you should really see what interests *you* and explore those directions further. Share and discuss any interesting findings in class and on Mattermost!
- Please don't stop reading at "Bonus"; see above. Don't be intimidated by all the text; pick something that interests you/lies within your capabilities and *just spend some time on it*.

Setting Up

To work on your own machine

Install Python (https://www.python.org/) (3.x -- depending on your OS you might need to install a not-so-recent version as the newest ones may be incompatible with TF) if you haven't done so, and install Tensorflow 2 (https://www.tensorflow.org/install/). This should be as simple as writing

{% highlight bash %} pip install tensorflow {% endhighlight %}

in your console. If you want GPU support (and have an appropriate GPU), you will need to follow extra steps (see the website). For now, there should be no need since you will usually use your own machine only for development and small tests.

If you want to do everything in Colab (see below), you don't need to install Tensorflow yourself.

Google Colab

Google Colab (https://colab.research.google.com) is a platform to facilitate teaching of machine learning/deep learning. There are tutorials available on-site. Essentially, it is a Jupyter notebook environment with GPU-supported Tensorflow available.

If you want to, you can develop your assignments within this environment. See below for some notes. Notebooks support Markup, so you can also write some text about what your code does, your observations etc. This is a really good idea!

Running code on Colab should be fairly straightforward; there are tutorials available in case you are not familiar with notebooks. There are just some caveats:

- You can check which TF version you are running via `tf.__version__` . Make sure this is 2.x!

- You will need to get external code (like `datasets.py`, see below) in there somehow. One option would be to simply copy and paste the code into the notebook so that you have it locally available. Another would be to run a cell with `from google.colab import files; files.upload()` and choose the corresponding file, this will load it "into the runtime" to allow you to e.g. import from it. Unfortunately you will need to redo this every time the runtime is restarted.
- Later you will need to make data available as well. Since the above method results in temporary files, the best option seems to be to upload them to Google Drive and use `from google.colab import drive; drive.mount('/content/drive')`. You might need to "authenticate" which can be a bit fiddly. After you succeed, you have your drive files available like a "normal" file system. If you find better ways to do this (or the above point), please share them with the class!
- In newer Colab versions, there is actually a button in the "Files" tab to mount the drive -- should be a bit simpler than importing `drive`. The following should work now:

    1. Find the folder in your Google Drive where the notebook is stored, by default this should be `Colab Notebooks`.
    2. Put your data, code (like `datasets.py` linked in one of the tutorials further below) etc. into the same folder (feel free to employ a more sophisticated file structure, but this folder should be your "root").
    3. Mount the drive via the button, it should be mounted into `/content`.
    4. Your working directory should be `content`, verify this via `os.getcwd()`.
    5. Use `os.chdir` to change your working directory to where the notebook is (and the other files as well, see step 2), e.g. `/content/drive/My Drive/Colab Notebooks`.
    6. You should now be able to do stuff like `from datasets import MNISTDataset` in your notebook (see MNIST tutorial further below).

Tensorflow Basics

NOTE: The Tensorflow docs went through significant changes recently. In particular, most introductory articles were changed from using low-level interfaces to high-level ones (particularly Keras). We believe it's better to start with low-level interfaces that force you to program every step of building/training a model yourself. This way, you actually need to understand what is happening in the code. High-level interfaces do a lot of "magic" under the hood. We will proceed to these interfaces after you learn the basics.

Get started with Tensorflow. There are many tutorials on diverse topics on the website, as well as an API documentation (https://www.tensorflow.org/api_docs/python/tf). The following should suffice for now:

- Read up on tensors and operations (https://www.tensorflow.org/tutorials/customization/basics) to get a basic idea of the stuff we are working with (you can ignore the section on Datasets). For more info, check the guide (https://www.tensorflow.org/guide/tensor).
- A special and very important kind of tensor are [variables].
- Automatic differentiation (https://www.tensorflow.org/tutorials/customization/autodiff) is probably the most important concept in Tensorflow!
- See how to fit a simple linear model (https://www.tensorflow.org/guide/basic_training_loops). You may ignore the Keras stuff (second part).

Linear Model for MNIST

MNIST is a collection of handwritten digits and a popular (albeit by now trivialized) benchmark for image classification models. You will see it A LOT.

Go through this basic MNIST tutorial (http://blog.ai.ovgu.de/posts/jens/2019/002_tf20_basic_mnist/index.html) we wrote just for you! It's a logistic (softmax) regression "walkthrough" both in terms of concepts and code. You will of course be tempted to just copy this code; please make sure you understand what each line does.

Play around with the example code snippets. Change them around and see if you can predict what's going to happen. Make sure you understand what you're dealing with!

Building A Deep Model

If you followed the tutorial linked above, you have already built a linear classification model. Next, turn this into a *deep* model by adding a hidden layer between inputs and outputs. To do so, you will need to add an additional set of weights and biases (after having chosen a size for the layer) as well as an activation function.
There you go! You have created a Multilayer Perceptron. **Hint:** Initializing variables to 0 will not work for multilayer perceptrons. You need to initialize values randomly instead (e.g. random_uniform between -0.1 and 0.1). Why do you think this is the case?

Next, you should explore this model: Experiment with different hidden layer sizes, activation functions or weight initializations. See if you can make any observations on how changing these parameters affects the model's performance. Going to extremes can be very instructive here. Make some plots!

Also, reflect on the Tensorflow interface: If you followed the tutorials you were asked to, you have been using a very low-level approach to defining models as well as their training and evaluation. Which of these parts do you think should be wrapped in higher-level interfaces? Do you feel like you are forced to provide any redundant information when defining your model? Any features you are missing so far?

Bonus

There are numerous ways to explore your model some more. For one, you could add more hidden layers and see how this affects the model. You could also try your hand at some basic visualization and model inspection: For example, visualize some of the images your model classifies incorrectly. Can you find out *why* your model has trouble with these?

You may also have noticed that MNIST isn't a particularly interesting dataset -- even very simple models can reach very high accuracy and there isn't much "going on" in the images. Luckily, Zalando Research has developed Fashion MNIST (https://github.com/zalandoresearch/fashion-mnist). This is a more interesting dataset with the *exact same structure* as MNIST, meaning you can use it without changing anything about your code. You can get it by simply using `tf.keras.datasets.fashion_mnist` instead of regular MNIST. You can attempt pretty much all of the above suggestions for this dataset as well!

How to Hand In Your Assignment

In general, you should prepare a notebook (`.ipynb` file) with your solution. The notebook should contain sufficient outputs that show your code working, i.e. **we should not have to run your code to verify that you solved the task**. You can use markdown cells to add some text, e.g. to document interesting observations you made or problems you ran into.

- If you work on Colab, make sure to save your notebooks with outputs! Under Edit -> Notebook settings, make sure the box with "omit code cell output..." is
- *not** ticked.

You can form groups to do the assignments (up to three people). However, **each group member needs to upload the solution separately** (because the Moodle group feature is a little broken). At the very top of the notebook, include the names of all group members!

What to hand in this time

Assignment 2: Let the Tensors Board? & also `tf.data`

Visualizing the learning progress as well as the behavior of a deep model is extremely useful (if not necessary) for troubleshooting in case of unexpected outcomes (or just bad results). In this assignment, you will get to know TensorBoard, Tensorflow's built-in visualization suite, and use it to diagnose some common problems with training deep models. **Note:** TensorBoard seems to work best with Chrome-based browsers. Other browsers may take a *very* long time to load, or not display the data correctly.

Datasets

It should go without saying that loading numpy arrays and taking slices of these as batches (as we did in the last assignment) isn't a great way of providing data to the training algorithm. For example, what if we are working with a dataset that doesn't fit into memory?

The recommended way of handling datasets is via the `tf.data` module. Now is a good time to take some first steps with this module. Read the Programmer's Guide section on this. You can ignore the parts on high-level APIs as well as anything regarding TFRecords and `tf.Example` (we will get to these later) as well as specialized topics involving time series etc. If this is still too much text for you, here is a super short version that just covers building a dataset from numpy arrays (ignore the part where they use Keras ;)). For now, the main thing is that you understand how to do just that.
Then, try to adjust your MLP code so that it uses `tf.data` to provide minibatches instead of the class in `datasets.py`. Keep in mind that you should map the data into the [0,1] range (convert to float!) and convert the labels to `int32` (check the old `MNISTDataset` class for possible preprocessing)!

Here you can find a little notebook that displays some basic `tf.data` stuff (also for MNIST).

Note that the Tensorflow guide often uses the three operations `shuffle`, `batch` and `repeat`. Think about how the results differ when you change the order of these operations (there are six orderings in total). You can experiment with a simple `Dataset.range` dataset. What do you think is the most sensible order?

First Steps with TensorBoard

As before, you will need to do some extra reading to learn how to use TensorBoard. There are several tutorials on the Tensorflow website, accessed via Resources -> Tools. However, they use many high-level concepts we haven't looked at yet to build their networks, so you can find the basics here. This is a modified version of last week's linear model that includes some lines to do TensorBoard visualizations. It should suffice for now. Integrate these lines into your MLP from the last assignment to make sure you get it to work! Basic steps are just:

- Set up a file writer for some log directory.

- During training, run summary ops for anything you are interested in, e.g.
  - Usually scalars for loss and other metrics (e.g. accuracy).
  - Distributions/histograms of layer activations or weights.
  - Images that show what the data looks like.
- Run TensorBoard on the log directory.

Later, we will also see how to use TensorBoard to visualize the *computation graph* of a model.

Finally, check out the github readme for more information on how to use the TensorBoard app itself (first part of the "Usage" section is outdated -- this is not how you create a file writer anymore).

Note: You don't need to hand in any of the above -- just make sure you get TensorBoard to work.

Diagnosing Problems via Visualization

Download this archive containing a few Python scripts. All these contain simple MLP training scripts for MNIST. All of them should also fail at training. For each example, find out through visualization why this is. Also, try to propose fixes for these issues. You may want to write summaries *every* training step. Normally this would be too much (and slow down your program), however it can be useful for debugging.

- These scripts/models are relatively simple -- you should be able to run them on your local machine as long as it's not too ancient. Of course you will need to have the necessary libraries installed.
- Please don't mess with the parameters of the network or learning algorithm before experiencing the original. You can of course use any oddities you notice as clues as to what might be going wrong.
- Sometimes it can be useful to have a look at the inputs your model actually receives. `tf.summary.image` helps here. Note that you need to reshape the inputs from vectors to 28x28-sized images and add an extra axis for the color channel (despite there being only one channel). Check out `tf.reshape` and `tf.expand_dims`.
- Otherwise, it should be helpful to visualize histograms/distributions of layer activations and see if anything jumps out. Note that histogram summaries will crash your program in case of `nan` values appearing. In this case, see if you can do without the histograms and use other means to find out what is going wrong.
- You should also look at the *gradients* of the network; if these are "unusual" (i.e. extremely small or large), something is probably wrong. An overall impression of a gradient's size can be gained via `tf.norm(g)`; feel free to add scalar summaries of these values to TensorBoard. You can pass a `name` to the variables when defining them and use this to give descriptive names to your summaries.
- Some things to watch out for in the code: Are the activation functions sensible? What about the weight initializations? Do the inputs/data look "normal"?
- Note: The final two scripts (4 and 5) may actually work somewhat, but performance should still be significantly below what could be achieved by a "correct" implementation.

What to Hand In

- An MLP training script using `tf.data`.
- A description of what the six `shuffle/batch/repeat` orderings do (on a conceptual level) and which one you think is the most sensible for training neural networks.
- For each "failed" script above, a description of the problem as well as how to fix it (there may be multiple ways). You can just write some text here (markdown cells!), but feel free to reinforce your ideas with some code snippets.
- Anything else you feel like doing. :)

Note that you can only upload a single notebook on Moodle, so use the Markdown features (text cells instead of code) to answer the text-based questions.

Bonus

Like last week, play around with the parameters of your networks. Use Tensorboard to get more information about how some of your choices affect behavior. For example, you could compare the long-term behavior of saturating functions such as *tanh* with *relu*, how the gradients vary for different architectures etc.

If you want to get deeper into the data processing side of things, check the Performance Guide.

Peer Quizzes Follow the registration instructions provided on the theory exercise channel on Mattermost. Contribute PeerQuiz platform with at least

- one question related to the course material and
- one question related to the programming assignment.

Answer and rate as many questions that are posted from peers as you want.

Assignment 3: Keras & CNNs

In this assignment, you will get to know Keras, the high-level API in Tensorflow. You will also create a better model for the MNIST dataset using convolutional neural networks.

Keras

The low-level TF functions we used so far are nice to have full control over everything that is happening, but they are cumbersome to use when we just need "everyday" neural network functionality. For such cases, Tensorflow has integrated Keras to provide abstractions over many common workflows. Keras has *tons* of stuff in it; we will only look at some of it for this assignment and get to know more over the course of the semester. In particular:

- `tf.keras.layers` provides wrappers for many common layers such as dense (fully connected) or convolutional layers. This takes care of creating and storing weights, applying activation functions, regularizers etc.
- `tf.keras.Model` in turn wraps layers into a cohesive whole that allows us to handle whole networks at once.
- `tf.optimizers` make training procedures such as gradient descent simple.
- `tf.losses` and `tf.metrics` allow for easier tracking of model performance.

Unfortunately, none of the TF tutorials are *quite* what we would like here, so you'll have to mix-and-match a little bit:

- This tutorial covers most of what we need, i.e. defining a model and using it in a custom training loop, along with optimizers, metrics etc. You can skip the part about GANs. Overall, the loop works much the same as before, except:
  - You now have all model weights conveniently in one place.
  - You can use the built-in loss functions, which are somewhat less verbose than `tf.nn.sparse_softmax_cross_entropy_with_logits`.
  - You can use `Optimizer` instances instead of manually subtracting gradients from variables.
  - You can use `metrics` to keep track of model performance.
- There are several ways to build Keras models, the simplest one being `Sequential`. For additional examples, you can look at the top of this tutorial, or this one, or maybe this one... In each, look for the part `model = tf.keras.Sequential...`. You just put in a list of layers that will be applied in sequence. Check the API to get an impression of what layers there are and which arguments they take.

Later, we will see how to wrap entire model definitions, training loops and evaluations in a hand-full of lines of code. For now, you might want to rewrite your MLP code with these Keras functions and make sure it still works as before.

An example notebook can be found here.

CNN for MNIST

You should have seen that (with Keras) modifying layer sizes, changing activation functions etc. is simple: You can generally change parts of the model without affecting the rest of the program (training loop etc). In fact, you can change *the full pipeline from input to model output* without having to change anything else (restrictions apply).

Replace your MNIST MLP by a CNN. The tutorials linked above might give you some ideas for architectures. Generally:

- Your data needs to be in the format `width x height x channels`. So for MNIST, make sure your images have shape `(28, 28, 1)`, not `(784,)`!
- Apply a bunch of `Conv2D` and possibly `MaxPool2D` layers.
- `Flatten`.
- Apply any number of `Dense` layers and the final classification (logits) layer.
- Use Keras!
- A reference CNN implementation *without* Keras can be found here!

  Note: Depending on your machine, training a CNN may take *much* longer than the MLPs we've seen so far. Here, using Colab's GPU support could be useful **(Edit -> Notebook settings -> Hardware Accelerator)**. Also, processing the full test set in one go for evaluation might be too much for your RAM. In that case, you could break up the test set into smaller chunks and average the results (easy using keras metrics) -- or just make the model smaller.

You should consider using a better optimization algorithm than the basic `SGD`. One option is to use adaptive algorithms, the most popular of which is called Adam. Check out `tf.optimizers.Adam`. This will usually lead to much faster learning without manual tuning of the learning rate or other parameters. We will discuss advanced optimization strategies later in the class, but the basic idea behind Adam is that it automatically chooses/adapts a per-parameter learning rate as well as incorporating momentum. Using Adam, your CNN should beat your MLP after only a few hundred steps of training. The general consensus is that a well-tuned gradient descent with momentum and learning rate decay will outperform adaptive methods, but you will need to invest some time into finding a good parameter setting -- we will look into these topics later.

If your CNN is set up well, you should reach extremely high accuracy results. This is arguably where MNIST stops being interesting. If you haven't done so, consider working with Fashion-MNIST instead (see Assignment 1). This should present more of a challenge and make improvements due to hyperparameter tuning more obvious/meaningful. You could even try CIFAR10 or CIFAR100 as in one of the tutorials linked above. They have 32x32 3-channel color images with much more variation. These datasets are also available in `tf.keras.datasets`. **Note:** For some reason, the CIFAR labels are organized somewhat differently -- shaped `(n, 1)` instead of just `(n,)`. You should do something like `labels = labels.reshape((-1,))` or this will mess up the loss function.

What to Hand In

- A CNN (built with Keras) trained on MNIST (or not, see below). Also use Keras losses, optimizers and metrics, but do still use a "custom" training loop (with `GradientTape`).

- You are *highly* encouraged to move past MNIST at this point. E.g. switching to CIFAR takes minimal effort since it can also be downloaded through Keras. You can still use MNIST as a "sanity check" that your model is working, but you can skip it for the submission.
- Document any experiments you try. For example:
  - Really do play with the model parameters. As a silly example, you could try increasing your filter sizes up to the input image size -- think about what kind of network you are ending up with if you do this! On the other extreme, what about 1x1 filters?
    - You can do the same thing for pooling. Or replace pooling with strided convolutions. Or...
  - If you're bored, just try to achieve as high of a (test set) performance as you can on CIFAR. This dataset is still commonly used as a benchmark today. Can you get ~97% (test set)?
  - You could try to "look into" your trained models. E.g. the convolutional layers output "feature maps" that can also be interpreted as images (and thus plotted one image per filter). You could use this to try to figure out what features/patterns the filters are recognizing by seeing for what inputs they are most active.

Assignment 4: Graphs & DenseNets

Note: Find the notebook from the exercises here.

Graph-based Execution

So far, we have been using so-called "eager execution" exclusively: Commands are run as they are defined, i.e. writing `y = tf.matmul(X, w)` actually executes the matrix multiplication.

In Tensorflow 1.x, things used to be different: Lines like the above would only *define the computation graph* but not do any actual computation. This would be done later in dedicated "sessions" that execute the graph. Later, eager execution was added as an alternative way of writing programs and is now the default, mainly because it is much more intuitive/allows for a more natural workflow when designing/testing models.

Graph execution has one big advantage: It is very efficient because entire models (or even training loops) can be executed in low-level C/CUDA code without ever going "back up" to Python (which is slow). As such, TF 2.0 still retains the possibility to run stuff in graph mode if you so wish -- let's have a look!

As expected, there is a tutorial on the TF website as well as this one which goes intro extreme depth on all the subtleties. The basic gist is:

- You can annotate a Python function with `@tf.function` to "activate" graph execution for this function.
- The first time this function is called, it will be *traced* and converted to a graph.
- Any other time this function is called, *the Python function will not be run; instead the traced graph is executed*.
- The above is not entirely true -- functions may be *retraced* under certain (important) conditions, e.g. for every new "input signature". This is treated in detail in the article linked above.
- Beware of using Python statements like `print`, these will not be traced so the statement will only be called during the tracing run itself. If you want to print things like tensor values, use `tf.print` instead. Basically, traced TF functions only do "tensor stuff", not general "Python stuff".

Go back to some of your pevious models and sprinkle some `tf.function` annotations in there. You might need to refactor slightly -- you need to actually wrap things into a function!

- The most straightforward target for decoration is a "training step" function that takes a batch of inputs and labels, runs the model, computes the loss and the gradients and applies them.
- In theory, you could wrap a whole training loop (including iteration over a dataset) with a `tf.function`. If you can get this to work on one of your previous models *and actually get a speedup*, you get a cookie. :)

DenseNet

Previously, we saw how to build neural networks in a purely sequential manner -- each layer receives one input and produces one output that serves as input to the next layer. There are many architectures that do not follow this simple scheme. You might ask yourself how this can be done in Keras. One answer is via the so-called functional API. There is an in-depth guide here. Reading just the intro should be enough for a basic grasp on how to use it, but of course you can read more if you wish.

Next, use the functional API to implement a DenseNet. You do *not* need to follow the exact same architecture, in fact you will probably want to make it smaller for efficiency reasons. Just make sure you have one or more "dense blocks" with multiple layers (say, three or more) each. You can also leave out batch normalization (this will be treated later in the class) as well as "bottleneck layers" (1x1 convolutions) if you want.

Bonus: Can you implement DenseNet with the Sequential API? You might want to look at how to implement custom layers (shorter version here)...

What to Hand In

- DenseNet. Thoroughly experiment with (hyper)parameters. Try to achieve the best performance you can on CIFAR10/100.
- For your model(s), compare performance with and without `tf.function`. You can also do this for non-DenseNet models. How does the impact depend on the size of the models?

The next two parts are just here for completeness/reference, to show other ways of working with Keras and some additional TensorBoard functionalities. Check them out if you want -- we will also (try to) properly present them in the exercise later.

Bonus: High-level Training Loops with Keras

As mentioned previously, Keras actually has ways of packing entire training loops into very few lines of code. This is good whenever you have a fairly "standard" task that doesn't require much besides iterating over a dataset and computing a loss/gradients at each step. In this case, you don't need the customizability that writing your own training loops gives you.

As usual, here are some tutorials that cover this:

- The gist is covered in the beginner quickstart: Build the model, compile with an optimizer, a loss and optional metrics and then run `fit` on a dataset. That's it!
- They also have the same thing with a bit more detail.
- The above covers the bare essentials, but you could also look at how to build a CNN for CIFAR10.

There are also some interesting overview articles in the "guide" section but this should suffice for now. Once again, go back to your previous models and re-make them with these high-level training loops! Also, from now on, feel free to run your models like this if you want (and can get it to work for your specific case).

Bonus: TensorBoard Computation Graphs

You can display the computation graphs Tensorflow uses internally in TensorBoard. This can be useful for debugging purposes as well as to get an impression what is going on "under the hood" in your models. More importantly, this can be combined with *profiling* that lets you see how much time/memory specific parts of your model take.

To look at computation graphs, you need to *trace* computations explicitly. See the last part of this guide for how to trace `tf.function` -annotated computations. Note: It seems like you have to do the trace the first time the function is called (e.g. on the first training step).

Assignment 5: Text Classification with RNNs (Part 1)

In this assignment and the next, we are switching to a different modality of data: Text. Namely, we will see how to assign a single label to input sequences of arbitrary length. This has many applications, such as detecting hate speech on social media or detecting spam emails. Here, we will look at sentiment analysis, which is supposed to tell what kind of emotion is associated with a piece of text.

In part 1, we are mainly concerned with implementing RNNs at the low level so that we understand how they work in detail. The models themselves will be rather rudimentary. We will also see the kinds of problems that arise when working with sequence data, specifically text. Next week, we will build better models and deal with some of these issues.

The notebook associated with the practical exercise can be found here.

The Data

We will be using the IMDB movie review dataset. This dataset comes with Keras and consists of 50,000 movie reviews with binary labels (positive or negative), divided into training and testing sets of 25,000 sequences each.

A first look

The data can be loaded the same way as MNIST or CIFAR -- `tf.keras.datasets.imdb.load_data()` . If you print the sequences, however, you will see that they are numbers, not text. Recall that deep learning is essentially a pile of linear algebra. As such, neural networks cannot take text as input, which is why it needs to be converted to numbers. This has already been done for us -- each word has been replaced by a number, and thus a movie review is a sequence of numbers (punctuation has been removed).

If you want to restore the text, `tf.keras.datasets.imdb.get_word_index()` has the mapping -- see the notebook for how you can use this, as well as some additional steps you need to actually get correct outputs.

Representing words

Our sequences are numbers, so they can be put into a neural network. But does this make sense? Recall the kind of transformations a layer implements: A linear map followed by a (optional) non-linearity. But that would mean, for example, that the word represented by index 10 would be "10 times as much" as the word represented by index 1. And if we simply swapped the mapping (which we can do, as it is completely arbitrary), the roles would be reversed! Clearly, this does not make sense.

A simple fix is to use one-hot vectors: Replace a word index by a vector with as many entries as there are words in the vocabulary, where all entries are 0 except the one corresponding to the

respective word, which is 1 -- see the notebook.

Thus, each word gets its own "feature dimension" and can be transformed separately. With this transformation, our data points are now sequences of one-hot vectors, with shape `(sequence_length, vocabulary_size)`.

Variable sequence lengths

Of course, not all movie reviews have the same length. This actually represents a huge problem for us: We would like to process inputs in batches, but tensors generally have to be "rectangular", i.e. we cannot have different sequence lengths in the same batch! The standard way to deal with this is *padding*: Appending additional elements to shorter sequences such that all sequences have the same length.

In the notebook, this is done in a rather crude way: All sequences are padded to the length of the longest sequence in the dataset.

Food for thought #1: Why is this wasteful? Can you think of a smarter padding scheme that is more efficient? Consider the fact that RNNs can work on arbitrary sequence lengths, and that training minibatches are pretty much independent of each other.

Dealing with extremes

Once we define the model, we will run into two issues with our data:

1. Some sequences are very long. This increases our computation time as well as massively hampering gradient flow. It is highly recommended that you limit the sequence length (200 could be a good start). You have two choices:
   A. *Truncate* sequences by cutting off all words beyond a limit. Both `load_data` and `pad_sequences` have arguments to do this. We recommend the latter as you can choose between "pre" or "post" truncation.
   B. *Remove* all sequences that are longer than a limit from the dataset. Radical!
2. Our vocabulary is large, more than 85,000 words. Many of these are rare words which only appear a few times. There are two reasons why this is problematic:

   A. The one-hot vectors are huge, slowing down the program and eating memory.
   B. It's difficult for the network to learn useful features for the rare words.

   `load_data` has an argument to keep only the `n` most common words and replace less frequent ones by a special "unknown word" token (index 2 by default). As a start, try keeping only the 20,000 most common words or so.

Food for thought #2: Between truncating long sequences and removing them, which option do you think is better? Why?

Food for thought #3: Can you think of a way to avoid the one-hot vectors completely? Even if you cannot implement it, a conceptual idea is fine.

With these issues taken care of, we should be ready to build an RNN!

Building The Model

A Tensorflow RNN "layer" can be confusing due to its black box character: All computations over a full sequence of inputs are done internally. **To make sure you understand how an RNN "works", you are asked to implement one from the ground up, defining variables yourself and using basic operations such as `tf.matmul` to define the computations at each time step and over a full input sequence.** There are some related tutorials available on the TF website, but all of these use Keras.

For this assignment, you are asked **not** to use the `RNNCell` classes nor any related Keras functionality. Instead, you should study the basic RNN equations and "just" translate these into code. You can still use Keras optimizers, losses etc. **You can also use `Dense` layers instead of low-level ops, but make sure you know what you are doing.** You might want to proceed as follows:

- On a high level, **nothing about the training loop changes!** The RNN gets an input and computes an output. The loss is computed based on the difference between outputs and targets, and gradients are computed and applied to the RNN weights, with the loss being backpropagated *trough time*.
- The differences come in how the RNN computes its output. The basic recurrency can be seen in equation 10.5 of the deep learning book, with more details in equations 10.8-10.11. The important idea is that, at each time step, the RNN essentially works like an MLP with a single hidden layer, but two inputs (last state and current input). In total, you need to "just":
  - Loop over the input, at each time step taking the respective slice. Your per-step input should be `batch x features` just like with an MLP!
  - At each time step, compute the new state based on the previous state as well as the current input.
  - Compute the per-step output based on the new state.
- What about comparing outputs to targets? Our targets are simple binary labels. On the other hand, we have one output *per time step*. The usual approach is to discard all outputs except the one for the very last step. Thus, this is a "many-to-one" RNN (compare figure 10.5 in the book).
- For the output and loss, you actually have two options:
  1. You could have an output layer with 2 units, and use sparse categorical cross-entropy as before (i.e. softmax activation). Here, whichever output is higher "wins".
  2. You can have *a single* output unit and use binary cross-entropy (i.e. sigmoid activation). Here, the output is usually thresholded at 0.5.

Food for thought #4: How can it be that we can *choose* how many outputs we have, i.e. how can both be correct? Are there differences between both choices as well as (dis)advantages relative to each other?

Open Problems

Initial state

To compute the state at the first time step, you would need a "previous state", but there is none. To fix this, you can define an "initial state" for the network. A common solution is to simply use a

tensor filled with zeros. You could also add a trainable variable and learn an initial state instead!

Food for thought #5: All sequences start with the same special "beginning of sequence" token (coded by index 1). Given this fact, is there a point in learning an initial state? Why (not)?

Computations on padded time steps

Recall that we padded all sequences to be the same length. Unfortunately, the RNN is not aware that we did this. This can be an issue, as we are basically computing new states (thus computing outputs as well as influencing future states) based on "garbage" inputs.

Food for thought #6: `pad_sequences` allows for `pre` or `post` padding. Try both to see the difference. Which option do you think is better? Recall that we use the final time step output from our model.

Food for thought #7: Can you think of a way to prevent the RNN from computing new states on padded time steps? One idea might be to "pass through" the previous state in case the current time step is padding. Note that, within a batch, some sequences might be padded for a given time step while others are not.

Slow learning

Be aware that it might take several thousand steps for the loss to start moving at all, so don't stop training too early if nothing is happening. Experiment with weight initializations and learning rates. For fast learning, the goal is usually to set them as large as possible without the model "exploding".

A major issue with our "last output summarizes the sequence" approach is that the information from the end has to backpropagate all the way to the early time steps, which leads to extreme vanishing gradient issues. You could try to use the RNN output more effectively. Here are some ideas:

- Instead of only using the final output, average (or sum?) the logits (pre-sigmoid) of all time steps and use this as the output instead.
- Instead of the logits, average the *states* at all time steps and compute the output based on this average state. Is this different from the above option?
- Compute logits and sigmoids for each output, and average the per-step probabilities.

Food for thought #8: What could be the advantage of using methods like the above? What are disadvantages? Can you think of other methods to incorporate the full output sequence instead of just the final step?

What to hand in

- A low-level RNN implementation for sentiment classification. If you can get it to move away from 50% accuracy on the training set, that's a success. Be wary of overfitting, however, as this doesn't mean that the model is generalizing! If the test (or validation) loss isn't moving, try using a smaller network. Also note that you may sometimes get a higher test accuracy, while the test loss is *also* increasing (how can this be?)!

- Consider the various questions posed throughout the assignment and try to answer them! You can use text cells to leave short answers in your notebook.

Assignment 6: Text Classification with RNNs (Part 2)

Building on the last assignment, this time we want to iron out some of the issues that were left. In particular:

- less wasteful padding,
- using embeddings to avoid one-hot vectors,
- avoiding computations on padded time steps,
- using Keras to simplify our code,
- using more advanced architectures such as LSTMs, stacked RNNs or bidirectional RNNs.

The notebook associated with the practical exercise can be found here.

Improving training efficiency

Within-batch padding

In the last assignment, we padded all sequences to the longest one in the dataset because we need "rectangular" input tensors. However, at the end of the day, only each *batch* of inputs needs to be the same length. If the longest sequence in a batch has length 150, the other sequences need only be padded to that length, not the longest in the whole dataset!

Thus, if we can find a way to delay the padding after we have formed batches, we can gain some efficiency. Unfortunately, we cannot even create a `tf.data.Dataset.from_tensor_slices` to apply batching to!

Luckily, there are other ways to create datasets. We will be using `from_generator`, which allows for creating datasets from elements returned by arbitrary Python generators. Even better, there is also a `padded_batch` transformation function which batches inputs *and* pads them to the longest length in the batch (what would happen if we tried the regular `batch` method?). See the notebook for a usage example!

Note: Tensorflow also has `RaggedTensor`. These are special tensors allowing different shapes per element. You can find a guide here. You could directly create a dataset `from_tensor_slices` by supplying a ragged tensor, which is arguably easier than using a generator. Unfortunately, ragged tensors are not supported by `padded_batch`. Sad! **However**, many tensorflow operations support ragged tensors, so padding can become unnecessary in many places! You can check the guide for an example with a Keras model. You can try this approach if you want, but for the rest of the assignment we will continue with the padded batches (the ragged version will likely be very slow).

Level 2: Bucketing

There is still a problem with the above approach. In our dataset, there are many short sequences and few very long ones. Unfortunately, it is very likely that all (or most) batches contain at least

one rather long sequence. That means that all the other (short) sequences have to be padded to the long one! Thus, in the worst case, our per-batch padding might only gain us very little. It would be great if there was a way to sort the data such that only sequences of a similar length are grouped in a batch... Maybe there is something in the notebook?

Note: If you truncated sequences to a relatively small value, like 200, bucketing may provide little benefit. The reason is that there will be so many sequences at the exact length 200 that the majority of batches will belong to this bucket. However, if you decide to allow a larger value, say length 500, bucketing should become more and more effective (noticeable via shorter time spent per batch).

Embeddings

Previously, we represented words by one-hot vectors. This is wasteful in terms of memory, and also the matrix products in our deep models will be very inefficient. It turns out, multiplying a matrix with a one-hot vector simply *extracts the corresponding column from the matrix*.

Keras offers an `Embedding` layer for an efficient implementation. Use this instead of the one-hot operation! Note that the layer adds additional parameters, however it can actually result in *fewer* parameters overall if you choose a small enough embedding size (recall the lecture discussion on using linear hidden layers).

RNNs in Keras

Keras offers various RNN layers. These layers take an entire 3d batch of inputs (`batch x time x features`) and return either a complete output sequence, or only the final output time step. There are two ways to use RNNs:

1. The more general is to define a *cell* which implements the per-step computations, i.e. how to compute a new state given a previous state and current input. There are pre-built cells for simple RNNs, GRUs and LSTMs (`LSTMCell` etc.). The cells are then put into the `RNN` layer which wraps them in a loop over time.
2. There also complete classes like `LSTM` which already wrap the corresponding cell.

While the first approach gives more flexibility (we could define our own cells), it is *highly* recommended that you stick with the second approach, as this provides highly optimized implementations for common usage scenarios. Check the docs for the conditions under which these implementations can be used!

Once you have an RNN layer, you can use ist just like other layers, e.g. in a sequential model. Maybe you have an embedding layer, followed by an LSTM, followed by a dense layer that produces the final output. Now, you can easily create stacked RNNs (just put multiple RNN layers one after the other), use `Bidirectional` RNNs, etc. Also try LSTMs vs GRUs!

Masking

One method to prevent new states being computed on padded time steps is by using a *mask*. A mask is a binary tensor with shape `(batch x time)` with 1s representing "real" time steps and 0s representing padding. Given such a mask, the state computation can be "corrected" like this:

```
new_state = mask_at_t * new_state + (1 - mask_at_t) * old_state
```

Where the mask is 1, the new state will be used. Where it is 0, the old state will be propagated instead!

Masking with Keras is almost too simple: Pass the argument `mask_zero=True` to your embedding layer (the constructor, not the call)! You can read more about masking here. The short version is that tensors can carry a mask as an attribute, and Keras layers can be configured to use and/or modify these masks in some way. Here, the embedding layer "knows" to create a mask such that 0 inputs (remember that index 0 encodes padding) are masked as `False`, and the RNN layers are implemented to perform something like the formula above.

Add masking to your model! The result should be much faster learning (in terms of steps needed to reach a particular performance, not time), in particular with `post` padding (the only kind of padding supported by `padded_batch`). The effect will be more dramatic the longer your sequences are.

What to hand in

Implement the various improvements outlined in this assignment. Experiment with adding them one by one and judge the impact (on accuracy, training time, convenience...) of each. You can also carry out "ablation" studies where you take the full model with all improvements, and remove them one at a time to judge their impact.

You can also try using higher or smaller vocabulary sizes and maximum sequence lengths and investigate the impact of these parameters!

Additional notes for custom RNN loops

If for some reason you are not using Keras RNN layers, but rather your own loops over time, there are a few more things to be aware of when using `tf.function`:

1. There seems to be an issue related to data shapes when using `bucket_by_sequence_length` and the final batch in the dataset (which can be smaller than the others). If you receive strange errors about unknown data shapes, you can set `drop_remainder=True`, or use regular `padded_batch` instead of bucketing.
2. A `tf.function` is re-compiled every time it receives an input with a different "signature". This is defined as the shape and data type of the tensor. When every batch has a different sequence length, this causes the training loop to be re-compiled every step. You can fix this by supplying an `input_signature` to `tf.function` -- please check the API docs. You can also pass `experimental_relax_shapes=True` instead, although this seems to be a little less effective.

Assignment 7: Attention-based Neural Machine Translation

In this task, you will implement a simple NMT with attention for a language pair of your choice. We will follow the corresponding TF Tutorial on NMT.

Please do **not** just use the exemplary English-Spanish example to reduce temptation of simply copying the tutorial.
You can find data sets here. We recommend picking a language pair where you understand both languages (so if you do speak Spanish... feel free to use it ;)). This makes it easier (and more fun) for you to evaluate the results. However, keep in mind that some language pairs have a very large amount of examples, whereas some only have very few, which will impact the learning process and the quality of the trained models.

You may run into issues with the code in two places:

1. The downloading of the data inside the notebook might not work (it crashes with a 403 Forbidden error). In that case, you can simply download & extract the data on your local machine and upload the .txt file to your drive, and then mount it and load the file as you've done before.
2. The `load_data` function might crash. It expects each line to result in *pairs of sentences*, but there seems to be a third element which talks about attribution of the example (at least if you download a different dataset from the link above). If this happens, you can use `line.split('\t')[:-1]` to exclude this in the function.

Tasks:

- Follow the tutorial and train the model on your chosen language pair.
- You might need to adapt the preprocessing depending on the language.
- Implement other attention mechanisms and train models with them (there are Keras layers for both):
  - Bahdanau attention (`AdditiveAttention`)
  - Luong's multiplicative attention (`Attention`)

Compare the attention weight plots for some examples between the attention mechanisms. We recommend to add `,vmax=1.0` when creating the plot in `ax.matshow(attention, cmap='viridis')` in the `plot_attention` function so the colors correspond to the same attention values in different plots.

- Do you see qualitative differences in the attention weights between different attention mechanisms?
- Do you think that the model attends to the correct tokens in the input language (if you understand both languages)?

Here are a few questions for you to check how well you understood the tutorial.
Please answer them (briefly) in your solution!

- Which parts of the sentence are used as a token? Each character, each word, or are some words split up?
- Do the same tokens in different language have the same ID?
  e.g. Would the same token index map to the German word `die` and to the English word `die`?
- Is the decoder attending to all previous positions, including the previous decoder predictions?
- Does the encoder output change in different decoding steps?
- Does the context vector change in different decoding steps?
- The decoder uses teacher forcing. Does this mean the time steps can be computed in parallel?
- Why is a mask applied to the loss function?
- When translating the same sentence multiple times, do you get the same result? Why (not)? If not, what changes need to be made to get the same result each time?

Hand in all of your code, i.e. the working tutorial code along with all changes/additions you made. Include outputs which document some of your experiments. Also remember to answer the questions above! Of course you can also write about other observations you made.

Assignment 8: Word2Vec

In this week, we will look at "the" classic model for learning word embeddings. This will be another tutorial-based assignment. Find the link here (https://www.tensorflow.org/tutorials/text/word2vec).

The key points are:

- Getting to know an example of *self-supervised learning*, where we have data without labels, and are constructing a task directly from the data (often some kind of prediction task) in order to learn deep representations,
- Understanding how Softmax with a very large number of classes is problematic, and getting to know possible workarounds,
- Exploring the idea of word embeddings.

Questions for Understanding

As in the last assignment, answer these questions in your submission to make sure you understand what is happening in the tutorial code!

- Given the sentence "I like to cuddle dogs", how many skipgrams are created with a window size of 2?
- In general, how does the number of skipgrams relate to the size of the dataset (in terms of input-target pairs)?
- Why is it not a good idea to compute the full softmax for classification?
- The way the dataset is created, for a given `(target, context)` pair, are the negative samples (remember, these are randomly sampled) the same each time this training example is seen, or are they different?
- For the given example dataset (Shakespeare), would the code create `(target, context)` pairs for sentences that span multiple lines? For example, the last word of one line and the first word of the next line?
- Does the code generate skipgrams for padding characters (index 0)?
- The `skipgrams` function uses a "sampling table". In the code, this is shown to be a simple list of probabilities, and it is created without any reference to the actual text data. How/why does this work? I.e. how does the program "know" which words to sample with which probability?

Possible Improvements & Extensions

- *If* the code generates skipgrams for padding characters: This is probably not a good idea. Can you prevent this from happening?
- *If* the code is not re-drawing negative samples each iteration: Can you change it so that it does? This may give less biased results.
- The candidate sampler may accidentally draw the true context word as one of the negative words. Can you find a way to detect and avoid this? Note that there is `tf.nn.sampled_softmax_loss` which supports such an argument. Using this would require significant re-writing of the code, however (e.g. getting rid of the `uniform_candidate_sampler` entirely).
- One of the most "impressive" features of these word embeddings is that, given a well-trained model, analogies can be performed via vector arithmetic. Try this:
  - Get the learned vectors (either target or context embeddings) for the words `king, queen, man, woman` . Of course, this assumes that these words are present in the training data.
  - Compute the vector `king - man + woman` .

    - Compute the similarity between the resulting vector and *all* word vectors. Which one gives the highest similarity? It "should" be `queen`. Note that it might actually be `king`, in which case `queen` should at least be second-highest. To compute the similarity, you should use cosine similarity.
    - You can try this for other pairs, such as `Paris - France + Germany = Berlin` etc.
- Use a larger vocabulary and/or larger text corpora to train the models. See how embedding quality and training effort changes. You can also implement a version using the "naive" full softmax, and see how the negative sampling increases in efficiency compared to the full version as the vocabulary becomes larger!

Optional: CBOW Model

The tutorial only covers the Skipgram model, however the same paper also proposed the (perhaps more intuitive) *Continuous Bag-of-Words* model. Here instead of predicting the context from the center word, it's the other way around. If you are looking for more of a challenge implementing a model by yourself, the changes should be as follows:

- In CBOW, each training example consists of *multiple* context words and a single target word. There is no equivalent to the `skipgrams` preprocessing function, but you can simply iterate over the full text data in small windows (there is `tf.data.Dataset.window` which may be helpful here) and for each window use the center word as the target and the rest as context.
- The context embedding is computed by embedding all context words separately, and then averaging their embeddings.

The rest stays pretty much the same. You will still need to generate negative examples through sampling, since the full softmax is just as inefficient as with the Skipgram model.

Compare the results of the CBOW model with the Skip

OPTIONAL ASSIGNMENT

Assignment 9: Self-Supervised Learning

In this assignment, we want to explore self-supervised methods for extracting features from unlabeled data, and then use those features for supervised tasks.

General Pipeline

No matter the exact kind of model, we usually do something like this:

1. Define a self-supervised task, such as autoencoding, denoising, predicting neighboring values, filling in blanks...
2. Build a network to solve the task. Often, this will be some kind of encoder-decoder architecture.
3. Train the model.
4. Build a small "classification head" on top of your self-supervised model. If that has an encoder-decoder structure, you will usually discard the decoder and put the classification head on top of the encoder.
5. Train the classification network on labeled data.

Your task

For a dataset of your choice, implement the above pipeline. Try **at least three different kinds** of self-supervised models; for each, train the model and then use the features for a classification task.
Also train a model directly on classification (no pre-training) and compare the performance to the self-supervised models. Also compare the different self-supervision methods with each other.

To make these comparisons fair, your models should have the same number of parameters. E.g. you might want to use the same "encoder" architecture for each task, and add a small classification head on top; then, the network that you train directly on classification should have the same architecture as the encoder and the classification head combined.

The remainder of this text discusses some issues to keep in mind when building autoencoders or similar models.

Autoencoders in Tensorflow

Building autencoders in Tensorflow is pretty simple. You need to define an encoding based on the input, a decoding based on the encoding, and a loss function that measures the distance between decoding and input. An obvious choice may be simply the mean squared error (but see below). To start off, you could try simple MLPs. Note that you are in no way obligated to choose the "reverse" encoder architecture for your encoder; e.g. you could use a 10-layer MLP as an encoder and a single layer as a decoder if you wanted. As a start, you should opt for an "undercomplete" architecture where the encoding is smaller than the data.

Note: The activation function of the last decoder layer is very important, as it needs to be able to map the input data range. Having data in the range [0, 1] allows you to use a sigmoid output activation, for example. Experiment with different activations such as sigmoid, relu or linear (i.e. no) activation and see how it affects the model. Your loss function should also "fit" the output function, e.g. a sigmoid output layer goes well with a binary (!) cross-entropy loss.

Note that you can use the Keras model APIs to build the encoder and decoder as different models, which makes it easy to later use the encoder separately. You can also have sub-models/layers participate in different models at the same time, e.g. an `encoder` model can be part of an `autoencoder` model together with a `decoder`, and of a classification model together with a `classifer_head`.

Convolutional Autoencoders

Next, you should switch to a convolutional encoder/decoder to make use of the fact that we are working with image data. The encoding should simply be one or more convolutional layers, with any filter size and number of filters (you can optionally apply fully-connected layers at the end). As an "inverse" of a `Conv2D`, `Conv2DTranspose` is commonly used. However, you could also use `UpSampling2D` along with regular convolutions. Again, there is no requirement to make the parameters of encoder and decoder "fit", e.g. you don't need to use the same filter sizes. However, you need to take care when choosing padding/strides such that the output has the same dimensions as the input. This can be a problem with MNIST (why?). It also means that the last convolutional (transpose) layer should have as many filters as the input space (e.g. one filter for MNIST or three for CIFAR).

Other models

Even other self-supervised models are often similar to autoencoders. For example, in a denoising autoencoder, the input is a noisy version of the target (so input and target are not the same anymore!), and the loss is computed between the output and this "clean" target. The architecture can remain the same, however.
Similarly, if the input has parts of the image removed and the task is to reconstruct those parts, the target is once again the full image, but an autoencoding architecture would in principle be appropriate once again.

To freeze or not to freeze

Say, you trained some encoder network on a self-supervised task and now build a classification head on top for labeled data. Now you want to train this model. But *which parts* do you actually train? It could be

- only the classification head, leaving the encoder untouched,
- the full network including the encoder,
- the classification head and some part of the encoder, say the last X layers...

There is a trade-off here:

- Allowing the encoder to be "fine-tuned" allows it to learn features that are suited for classification, in case the self-supervised features are not optimal.
- However, this might cause the encoder to overfit on the training set. Training only the classification head would keep the encoder features more general.
- The third option is a compromise between both.

Experiment! You can easily "freeze" layers or whole models by setting their `trainable` argument to `False`.

Pre-training for low supervision scenarios

Self-supervised models are useful in that they can learn from unlabeled data. This can significantly improve performance in settings where large amounts of data are available, but few labels. We can artificially evoke such a situation by just "pretending" that parts of our data has no labels. Try this:

- Train a self-supervised model as before.
- Take a small random subset of the training data. Make sure it is actually random, i.e. all labels are represented. You could go very low, e.g. 100 elements or so.
- Now train a classification net on only this small labeled subset!

As before, compare to a model that is trained directly on the classification task, but only on the labeled subset. If everything works as expected, your self-supervised model should significantly outperform the directly trained one (on the test set)! This is because the direct training massively overfits on the small dataset, whereas the self-supervised model was able to learn features on all available data. You will most likely want to freeze the encoder model, i.e. not fine-tune it -- if you did, the self-supervised model would overfit, as well.