For this year’s course edition, we created a series of Jupyter notebooks that are designed to help you understanding the “theory” from the lectures by seeing corresponding implementations. We will visit various topics such as optimization techniques, graph neural networks and adversarial attacks (for a full list, see below). The notebooks are there to help you understand the material and teach you details of the PyTorch framework, including PyTorch Lightning.

The notebooks are presented in the second hour of each lecture slot. During the tutorial sessions, we will present the content and explain the implementation of the notebooks. You can decide yourself rather you just want to look at the filled notebook, want to try it yourself, or code along during the practical session. We do not have any mandatory assignments on which you would be graded or similarly. However, we encourage you to get familiar with the notebooks and experiment or extend them yourself.
### CHAPTER ONE

### SCHEDULE

<table>
<thead>
<tr>
<th>Date</th>
<th>Notebook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thursday, 29. October 2020, 13.00-14.00</td>
<td>Tutorial 2: Introduction to PyTorch</td>
</tr>
<tr>
<td>Tuesday, 3. November 2020, 17.00-18.00</td>
<td>Tutorial 3: Activation functions</td>
</tr>
<tr>
<td>Thursday, 5. November 2020, 12.00-13.00</td>
<td>Tutorial 4: Optimization and Initialization</td>
</tr>
<tr>
<td>Tuesday, 10. November 2020, 14.00-15.00</td>
<td>Tutorial 5: Inception, ResNet and DenseNet</td>
</tr>
<tr>
<td>Thursday, 12. November 2020, 12.00-13.00</td>
<td>Tutorial 6: Transformers and Multi-Head Attention</td>
</tr>
<tr>
<td>Tuesday, 17. November 2020, 14.00-15.00</td>
<td>Guide 2: Research projects with PyTorch</td>
</tr>
<tr>
<td>Thursday, 19. November 2020, 12.00-13.00</td>
<td>Tutorial 7: Graph Neural Networks</td>
</tr>
<tr>
<td>Thursday, 26. November 2020, 12.00-13.00</td>
<td>Tutorial 9: Autoencoders</td>
</tr>
<tr>
<td>Tuesday, 1. December 2020, 14.00-15.00</td>
<td>Tutorial 10: Adversarial Attacks</td>
</tr>
<tr>
<td>Tuesday, 8. December 2020, 14.00-15.00</td>
<td>Tutorial 11: Normalizing Flows</td>
</tr>
<tr>
<td>Thursday, 10. December 2020, 12.00-13.00</td>
<td>Tutorial 12: Autoregressive Image Modeling</td>
</tr>
</tbody>
</table>
On this website, you will find the notebooks exported into a HTML format so that you can read them from whatever device you prefer. However, we suggest that you also give them a try and run them yourself. There are three main ways of running the notebooks we recommend:

- **Locally on CPU**: All notebooks are stored on the github repository that also builds this website. You can find them here: https://github.com/phlippe/uvadlc_notebooks/tree/master/docs/tutorial_notebooks. The notebooks are designed that you can execute them on common laptops without the necessity of a GPU. We provide pretrained models that are automatically downloaded when running the notebooks, or can manually be downloaded from this Google Drive. The required disk space for the pretrained models and datasets is less than 1GB. To ensure that you have all the right python packages installed, we provide a conda environment in the same repository.

- **Google Colab**: If you prefer to run the notebooks on a different platform than your own computer, or want to experiment with GPU support, we recommend using Google Colab. Each notebook on this documentation website has a badge with a link to open it on Google Colab. Remember to enable GPU support before running the notebook (Runtime -> Change runtime type). Each notebook can be executed independently, and doesn’t require you to connect your Google Drive or similar. However, when closing the session, changes might be lost if you don’t save it to your local computer or have copied the notebook to your Google Drive beforehand.

- **Lisa cluster**: If you want to train your own (larger) neural networks based on the notebooks, you can make use of the Lisa cluster. However, this is only suggested if you really want to train a new model, and use the other two options to go through the discussion and analysis of the models. Lisa might not allow you with your student account to run jupyter notebooks directly on the gpu_shared partition. Instead, you can first convert the notebooks to a script using jupyter nbconvert --to script ...ipynb, and then start a job on Lisa for running the script. A few advices when running on Lisa:
  - Disable the tqdm statements in the notebook. Otherwise your slurm output file might overflow and be several MB large. In PyTorch Lightning, you can do this by setting progress_bar_refresh_rate=0 in the trainer.
  - Comment out the matplotlib plotting statements, or change plt.show() to plt.savefig(...).
We will discuss 12 tutorials in total, each focusing on a different aspect of Deep Learning. The tutorials are spread across lectures, and we tried to cover something from every area. You can align the tutorials with the lectures as follows:

- **Lecture 1**: Introduction to Deep Learning
  - Guide 1: Working with the Lisa cluster
  - Tutorial 2: Introduction to PyTorch
- **Lecture 2**: Modular Learning
  - Tutorial 3: Activation functions
- **Lecture 3**: Deep Learning Optimizations
  - Tutorial 4: Optimization and Initialization
- **Lecture 4**: Convolutional Neural Networks
- **Lecture 5**: Modern ConvNets
  - Tutorial 5: Inception, ResNet and DenseNet
- **Lecture 6**: Recurrent Neural Networks
  - Tutorial 6: Transformers and Multi-Head Attention
- **Lecture 7**: Graph Neural Networks
  - Tutorial 7: Graph Neural Networks
- **Lecture 8**: Deep Generative Models
  - Tutorial 8: Deep Energy Models
- **Lecture 9**: Deep Variational Inference
  - Tutorial 9: Deep Autoencoders
- **Lecture 10**: Generative Adversarial Networks
  - Tutorial 10: Adversarial Attacks
- **Lecture 11**: Advanced Generative Models
  - Tutorial 11: Normalizing Flows
  - Tutorial 12: Autoregressive Image Modeling
- **Lecture 12**: Deep Stochastic Models
- **Lecture 13**: Bayesian Deep Learning
• Lecture 14: Deep Dynamics
This is the first time we present these tutorials during the Deep Learning course. As with any other project, small bugs and issues are expected. We appreciate any feedback from students, whether it is about a spelling mistake, implementation bug, or suggestions for improvements/additions to the notebooks. Please use the following link to submit feedback, or feel free to reach out to me directly per mail (p dot lippe at uva dot nl), or grab me during any TA session.
Not all tutorials have been finished yet, and some are still in the progress of being created. Below you can find an overview of the progress status. Each notebook has also a badge indicating its status (In progress, First version, Finished).

**Tutorials finished**

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

**Tutorials in work**

**Tutorials skipped this year**

- Tutorial 13: Bayesian Deep Learning
- Tutorial 14: Deep Dynamics

### 5.1 Guide 1: Working with the Lisa cluster

This tutorial explains how to work with the Lisa cluster for the Deep Learning course. It is recommended to have listened to the presentation by the SURFsara team before going through this tutorial.
5.1.1 First steps

How to connect to Lisa

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

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

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

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

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

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

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

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

Modules

Lisa uses modules to provide you various pre-installed software. This includes simple Python, but also the NVIDIA libraries CUDA and cuDNN that are necessary to use GPUs in PyTorch. A standard pack of software we use is the following:

```
module load 2019
module load Python/3.7.5-foss-2019b
module load CUDA/10.1.243
module load cuDNN/7.6.5.32-CUDA-10.1.243
module load NCCL/2.5.6-CUDA-10.1.243
```

When working on the login node, it is sufficient to load the 2019 software pack and the Python/... module. CUDA and cuDNN is only required when you run a job on a node.

Install the environment

To run the Deep Learning assignments and other code like the notebooks on Lisa, you need to install the provided environment for Lisa. Lisa provides an Anaconda module, which you can load via `module load Anaconda3/2018.12` (remember to load the 2019 module beforehand). Install the environment with the following command:

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

If you experience issues with the Anaconda module, you can also install Anaconda yourself (download link) or ask your TA for help.
5.1.2 The SLURM scheduler

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

Job files

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

```
#!/bin/bash
#SBATCH --partition=gpu_shared_course
#SBATCH --gres=gpu:1
#SBATCH --job-name=ExampleJob
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=3
#SBATCH --time=04:00:00
#SBATCH --mem=32000M
#SBATCH --output=slurm_output_%A.out
module purge
module load 2019
module load Python/3.7.5-foss-2019b
module load CUDA/10.1.243
module load cuDNN/7.6.5.32-CUDA-10.1.243
module load NCCL/2.5.6-CUDA-10.1.243
module load Anaconda3/2018.12

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

Job arguments

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

- **partition**: The partition of Lisa on which you want to run your job. As a student, you only have access to the partition `gpu_shared_course`, which provides you nodes with NVIDIA GTX1080Ti GPUs (11GB).

- **gres**: Generic resources include the GPU which is crucial for deep learning jobs. You can select up to two GPUs with your account, but if you haven’t designed your code to explicitly run on multiple GPUs, please use only one GPU (so no need to change what we have above).

- **job-name**: Name of the job to pop up when you list your jobs with `squeue` (see below).

- **ntasks**: Number of tasks to run with the job. In our case, we will always use 1 task.

- **cpus-per-task**: Number of CPUs you request from the nodes. The `gpu_shared_course` partition restricts you to max. 3 CPUs per job/GPU.

- **time**: Estimated time your job needs to finish. It is no problem if your job finishes earlier than the specified time. However, if your job takes longer, it will be instantaneously killed after the specified time. Still, don’t
specify unnecessarily long times as this causes your job to be scheduled later (you need to wait longer in the queue if other people also want to use the cluster). A good rule of thumb is to specify ~20% more than what you would expect.

- **mem**: RAM of the node you need. Note that this is not the GPU memory, but the random access memory of the node. On gpu_shared_course, you are restricted to 64GB per job/GPU which is more than you need for the assignments.

- **output**: Output file to which the slurm output should be written. The tag “%A” is automatically replaced by the job ID. Note that if you specify the output file to be in a directory that does not exist, no output file will be created.

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

### Scratch

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

### Starting and organizing jobs

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

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

- **squeue**: Lists all jobs that are currently submitted to Lisa. This can be a lot of jobs as it includes all partitions. You can make it partition-specific using `squeue -p gpu_shared_course`, or only list the jobs of your account: `squeue -u lgpu___` (again, replace `lgpu___` by your username). See the slurm documentation for details.

- **scancel JOBID**: Cancels and stops a job, independent of whether it is running or pending. The job ID can be found using `squeue`, and is printed when submitting the job via `sbatch`.

- **sinfo control show JOBID**: Shows additional information of a specific job, like the estimated start time.

### 5.1.3 Troubleshooting

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

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

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

If none of those work, the connection issue is likely not on your side. The problem usually resolves after 2-3 hours, and Lisa let’s you login after it again. If the problem doesn’t resolve after couple of hours, please contact your TA, and eventually the SURFSara helpdesk.

Slurm output file missing

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

Slurm output file is empty for a long time

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

All my jobs are pending

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

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

You can also see the estimated start time of a job by running `sinfo control show JOBID`. However, note that this is the “worst case” scenario for the current number of submitted jobs, as in if all currently running jobs would need their maximum runtime. At the same time, if more people would submit their job with higher priority, yours can fall back in the queue and get a later start time.
5.2 Guide 2: Research projects with PyTorch

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

5.2.1 Setup

Framework

- Choose the right framework. If you have simple setups like classification, consider going with PyTorch Lightning. If you need to change the default training procedure, go with plain PyTorch and write your own framework
- Usually a good setup:

```plaintext
general/
  |  train.py
  |  task.py
  |  mutils.py
layers/
experiments/
  |  task1/
  |    |  train.py
  |    |  task.py
  |    |  eval.py
  |    |  dataset.py
  |  task2/
  |    |  train.py
  |    |  task.py
  |    |  eval.py
  |    |  dataset.py
```

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

- It is a good practice to use argument parsers for specifying hyperparameters. Argument parsers allow you to call a training like `python train.py --learning ... --seed ... --hidden_size ...` etc.

- If you have multiple models to choose from, you will have multiple sets of hyperparameters. A good summary on that can be found in the PyTorch Lightning documentation without the need of using Lightning.

5.2.2 Hyperparameter search

- In general, hyperparameter search is all about experience. Once you have trained a lot of models, it will become easier for you to pick reasonable first-guess hyperparameters.

- Another good approach is to look at related work to your model, and see what others have used as hyperparameters for similar models. This will help you to get started with a reasonable choice.

- Hyperparameter search can be expensive. Thus, try to do the search on shallow models first before scaling them up.

- Although a large grid search is the best way to get the optimum out of your model, it is often not reasonable to run. Try to group hyperparameters, and optimize each group one by one.

Toolkits

- PyTorch Lightning provides a lot of useful tricks and toolkits, such as:
  - Learning rate finder that plots the learning rate vs loss for a few initial batches, and helps you to choose a reasonable learning rate.
  - Autoscaling batch sizes which finds the largest possible batch size given your GPU (helpful if you have very deep, large models, and it is obvious you need the largest batch size possible)

- For comparing multiple hyperparameter configurations, you can add them to TensorBoard. This is a clean way of comparing multiple runs. If interested, a blog on this can be found here

- There are multiple libraries that support you in automatic hyperparameter search. A good overview for those in PyTorch can be found here

Reproducibility

- Everything is about reproducibility. Make sure you can reproduce any training you do with the same random values, batches, etc. You will come to a point where you have tried a lot of different approaches, but none were able to improve upon one of your previous runs. When you try to run the model again with the best hyperparameters, you don’t want to have a bad surprise (believe me, enough people have this issue, and it can also happen to you). Hence, before starting any grid search, make sure you are able to reproduce runs. Run two jobs in parallel on Lisa with the same hyperparams, seeds, etc., and if you don’t get the exact same results, stop and try to fix it before anything else.

- Another fact about reproducibility is that saving and loading a model works without any problems. Make sure before a long training that you are able to load a saved model from the disk, and achieve the exact same test score as you had during training.

- Print your hyperparameters into the SLURM output file (simple print statement in python). This will help you identifying the runs, and you can easily check whether Lisa executes the job you intended to

- When running a job, copy the job file automatically to your checkpoint folder. Improves reproducibility
• Besides the slurm output file, create a output file in which you store the best training, validation and test score. This helps when you want to compare

Seeds

• DL models are noisy. Before running a grid search, try to get a feeling of how noisy your experiments might be. The more noise you expect compared to
• After finishing the grid search, run another model of the best configuration with a new seed. If the score is still the best, take the model. If not, consider running a few more seeds for the top $k$ models in your grid search. Otherwise you risk taking a suboptimal model, which was just lucky to the best for a specific seed.

Learning rate

• Depends on optimizer, model and many more other hyperparameters
• A usual good starting point for SGD is 0.1, and Adam 1e-3
• The deeper the model is, the lower the learning rate usually should be
• The lower your batch, the lower the lr should be. Consider using gradient accumulation if your batch size is getting too small (PyTorch Lightning supports this, see here).
• Consider using the PyTorch Lightning learning rate finder toolkit for an initial good guess.

LR scheduler

• It again depends on the classifier and model
• For classifiers and SGD, the multi-step LR has shown to be good
• Models trained with Adam commonly use a smooth exponential decay in the learning rate
• For Transformers: remember to use a learning rate warmup, the cosine scheduler is often used for decaying the learning rate afterwards

Regularization

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

- There are couple of regularization techniques that depend on your input data/domain. The most common include:
  - Computer Vision: image augmentation
  - NLP: input dropout of whole words
  - Graphs: dropping edges, inputs

Grid search with SLURM

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

Job arrays

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

File `array_job.job`:

```bash
#!/bin/bash

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

module purge
module load 2019
module load Python/3.7.5-foss-2019b
```

(continues on next page)
module load CUDA/10.1.243
module load cuDNN/7.6.5.32-CUDA-10.1.243
module load NCCL/2.5.6-CUDA-10.1.243
module load Anaconda3/2018.12

# Your job starts in the directory where you call sbatch

# Activate your environment
source activate ...

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

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

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

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

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

File array_job_hyperparameters.txt:

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

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

5.3 Guide 3: Debugging in PyTorch

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

5.3.1 My model is underperforming

These errors are the most “evil” and hard to debug. Your model is not reaching the performance that it should, but PyTorch is not telling you why that happens. There are couple of things you can check. If none of these solve the problem for you, one of us TAs will help you debug your code more in detail.

Softmax, CrossEntropyLoss and NLLLoss

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

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

Softmax over the right dimension

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

Categorical data and Embedding

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

1. You bias the model to see relations where there are none. In the language example above, the model would think that ‘a’ is closer to ‘b’ than to ‘o’, although ‘a’ and ‘o’ are both vocals, and the closeness of ‘a’ and ‘b’ does not necessarily say anything about there usage.

2. If you have many categories, you will have input values between 0 and >50. The model will have a hard time separating all those >50 categories without blending over some. Hence, the model loses a lot of information although this is not necessary.
The much better option in the case of categorical data is to use one-hot vectors, or embeddings. A one-hot vector represents each category by a vector of 0s, with one index being 1. This makes the model’s life much easier as it can distinguish between the categories in a very simple manner (if feature ≠ 0, it is a specific category). Alternatively, you can learn an embedding with the help of `nn.Embedding`. The inputs to this module are:

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

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

```python
[1]: import torch
   import torch.nn as nn

   # Create 5 embedding vectors each with 32 features
   embedding = nn.Embedding(num_embeddings=5,
                              embedding_dim=32)

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

   # Get embeddings
   embed_vectors = embedding(input_tensor)

   print("Input shape:", input_tensor.shape)
   print("Output shape:", embed_vectors.shape)
   print("Example features:

   tensor([[ 0.0504, -0.2422],
           [ 0.0342, 0.2217]],
      [ 2.7813, -0.3641],
      [-0.0981, 0.4069]],
      [ 0.0504, -0.2422],
      [ 1.2092, 0.1760]], grad_fn=<SliceBackward>)
```

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

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

Hidden shape mismatch

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

Training and Evaluation switch

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

Parameter handling

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

Parameters and `.to(device)`

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

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

Initialization

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

Zero-grad in optimizers

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

Weight decay and Adam

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

Check your metric calculation

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

My bits per dimension score is very low

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

5.3.2 PyTorch throws an error

These errors are the “happy” errors because PyTorch actually talks to you about what is wrong. Until these are not solved, you probably cannot train your model.
Trying to backward through the graph a second time, specify retain_graph=True

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

Size mismatch

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

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

Device mismatch

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

5.3.3 Good practices

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

Use nn.Sequential and nn.ModuleList

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

```python
[2]: class MLP(nn.Module):
    def __init__(self, input_dims=64, hidden_dims=[128,256], output_dims=10):
        super().__init__()
        hidden_dims = [input_dims] + hidden_dims
        layers = []
        for idx in range(len(hidden_dims)-1):
            layers += [
                nn.Linear(hidden_dims[i], hidden_dims[i+1]),
                nn.ReLU(inplace=True)
            ]
        self.layers = nn.Sequential(*layers)
    def forward(self, x):
        return self.layers(x)
```

5.3. Guide 3: Debugging in PyTorch
In-place activation functions

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

Create modules for repeating blocks

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

Stack layers/weights with same input

If you have multiple linear layers or convolutions that have the same input, you can stack them together to increase efficiency. Suppose we have two layers on `x`: `y1 = W1x + b1, y2 = W2x + b2`. While you could implement it by two linear layers, you can get the exact same neural network by stacking the two layers into one. The single layer is more efficient as this represents a single matrix operation instead of two for the GPU, and hence we can parallelize the computation. An example is shown below:

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

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

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

If you implement the linear layer manually, you need to stack the weight and bias tensor accordingly. Note that you should change your initialization for the stacked case if necessary. If your initialization depends on the output size of a layer (as for example in Xavier), you would get a different standard deviation for initialization in the two implementations. Still, if your initialization solely depends on the input dimension (e.g. Kaiming), no change is necessary. An example case where this stacking can be beneficial is LSTMs as all four gates use the exact same input.
Use loss functions on logits

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

Make use of torch.nn.functional

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

Clip gradient norms

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

5.4 Tutorial 2: Introduction to PyTorch

Filled notebook:

Empty notebook:

Welcome to our PyTorch tutorial for the Deep Learning course 2020 at the University of Amsterdam! The following notebook is meant to give a short introduction to PyTorch basics, and get you setup for writing your own neural networks. PyTorch is an open source machine learning framework that allows you to write your own neural networks and optimize them efficiently. However, PyTorch is not the only framework of its kind. Alternatives to PyTorch include TensorFlow, JAX and Caffe. We choose to teach PyTorch at the University of Amsterdam because it is well established, has a huge developer community (originally developed by Facebook), is very flexible and especially used in research. Many current papers publish their code in PyTorch, and thus it is good to be familiar with PyTorch as well. Meanwhile, TensorFlow (developed by Google) is usually known for being a production-grade deep learning library. Still, if you know one machine learning framework in depth, it is very easy to learn another one because many of them use the same concepts and ideas. For instance, TensorFlow’s version 2 was heavily inspired by the most popular features of PyTorch, making the frameworks even more similar. If you are already familiar with PyTorch and have created your own neural network projects, feel free to just skim this notebook.

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

We will use a set of standard libraries that are often used in machine learning projects. If you are running this notebook on Google Colab, all libraries should be pre-installed. If you are running this notebook locally, make sure you have installed our dl2020 environment and have activated it.
## Standard libraries

```python
import os
import math
import numpy as np
import time
```

## Imports for plotting

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

## Progress bar

```python
from tqdm.notebook import tqdm
```

### 5.4.1 The Basics of PyTorch

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

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

```python
import torch
print("Using torch", torch.__version__)
```

Using torch 1.6.0

At the time of writing this tutorial (mid of October 2020), the current stable version is 1.6. You should therefore see the output `Using torch 1.6.0`. If you see a lower version number, make sure you have installed the correct environment, or ask one of your TAs. In case PyTorch 1.7 or newer will be published during the time of the course, don’t worry. The interface between PyTorch versions doesn’t change too much, and hence all code should also be runnable with newer versions.

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

```python
torch.manual_seed(42)  # Setting the seed
```

```
torch._C.Generator at 0x7f785df1a480
```
Tensors

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

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

Initialization

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

```python
x = torch.Tensor(2, 3, 4)
print(x)
tensor([[1.0949e-32, 1.3563e-19, 1.7589e+22, 1.2412e+28],
        [6.6830e+22, 1.7199e+11, 4.8617e+30, 1.8499e+20],
        [1.5265e-19, 2.0702e-19, 1.0951e+21, 2.0339e+32]],
       [[1.9346e-19, 1.2711e+31, 1.9346e-19, 1.9432e-19],
        [4.5317e-11, 1.3563e-19, 2.1973e-18, 1.3563e-19],
        [6.7722e+22, 1.6020e-19, 2.6563e+20, 1.7613e+19]]])
```

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

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

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

# Create a tensor with random values between 0 and 1 with the shape [2, 3, 4]
x = torch.rand(2, 3, 4)
print(x)
tensor([[0.8823, 0.9150, 0.3829, 0.9593],
        [0.3904, 0.6009, 0.2566, 0.7936],
        [0.9408, 0.1332, 0.9346, 0.5936]],
       [[0.8694, 0.5677, 0.7411, 0.4294],
        [0.8854, 0.5739, 0.2666, 0.6274],
        [0.2696, 0.4414, 0.2969, 0.8317]])
```
You can obtain the shape of a tensor in the same way as in numpy (\texttt{x.shape}), or using the \texttt{.size} method:

```python
[7]: shape = x.shape
    print("Shape:", x.shape)

size = x.size()
    print("Size:", size)

dim1, dim2, dim3 = x.size()
    print("Size:", dim1, dim2, dim3)
```

Shape: torch.Size([2, 3, 4])
Size: torch.Size([2, 3, 4])
Size: 2 3 4

**Tensor to Numpy, and Numpy to Tensor**

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

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

print("Numpy array:", np_arr)
print("PyTorch tensor:", tensor)
```

Numpy array: [[1 2]
             [3 4]]
PyTorch tensor: tensor([[1, 2],
                       [3, 4]])

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

```python
[9]: tensor = torch.arange(4)
    np_arr = tensor.numpy()

print("PyTorch tensor:", tensor)
print("Numpy array:", np_arr)
```

PyTorch tensor: tensor([0, 1, 2, 3])
Numpy array: [0 1 2 3]

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

**Operations**

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

The simplest operation is to add two tensors:

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

(continues on next page)
print("X1", x1)
print("X2", x2)
print("Y", y)

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

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

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

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

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

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

```
[12]: x = torch.arange(6)
print("X", x)
X tensor([0, 1, 2, 3, 4, 5])

[13]: x = x.view(2, 3)
print("X", x)
X tensor([[0, 1, 2],
          [3, 4, 5]])

[14]: x = x.permute(1, 0) # Swapping dimension 0 and 1
print("X", x)
X tensor([[0, 3],
          [1, 4],
          [2, 5]])
```
Other commonly used operations include matrix multiplications, which are essential for neural networks. Quite often, we have an input vector \( \mathbf{x} \), which is transformed using a learned weight matrix \( \mathbf{W} \). There are multiple ways and functions to perform matrix multiplication, some of which we list below:

- **torch.matmul**: Performs the matrix product over two tensors, where the specific behavior depends on the dimensions. If both inputs are matrices (2-dimensional tensors), it performs the standard matrix product. For higher dimensional inputs, the function supports broadcasting (for details see the documentation). Can also be written as `\mathbf{a} @ \mathbf{b}`, similar to numpy.

- **torch.mm**: Performs the matrix product over two matrices, but doesn’t support broadcasting (see documentation)

- **torch.bmm**: Performs the matrix product with a support batch dimension. If the first tensor \( \mathbf{T} \) is of shape \((b \times n \times m)\), and the second tensor \( \mathbf{R} \) \((b \times m \times p)\), the output \( \mathbf{O} \) is of shape \((b \times n \times p)\), and has been calculated by performing \( b \) matrix multiplications of the submatrices of \( \mathbf{T} \) and \( \mathbf{R} \):
  \[
  O_i = T_i @ R_i
  \]

- **torch.einsum**: Performs matrix multiplications and more (i.e. sums of products) using the Einstein summation convention. Explanation of the Einstein sum can be found in assignment 1.

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

```python
[15]: x = torch.arange(6)
x = x.view(2, 3)
print("X", x)
X tensor([[0, 1, 2],
         [3, 4, 5]])

[16]: W = torch.arange(9).view(3, 3)# We can also stack multiple operations in a single-line
   print("W", W)
W tensor([[0, 1, 2],
         [3, 4, 5],
         [6, 7, 8]])

[17]: h = torch.matmul(x, W)# Verify the result by calculating it by hand too!
   print("h", h)
h tensor([[15, 18, 21],
         [42, 54, 66]])
```

### Indexing

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

```python
[18]: x = torch.arange(12).view(3, 4)
   print("X", x)
X tensor([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]])

[19]: print(x[:, 1])# Second column
tensor([1, 5, 9])
```
Dynamic Computation Graph and Backpropagation

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

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

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

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

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

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

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

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

```python
[24]:
x.requires_grad_(True)
print(x.requires_grad)
True
```
In order to get familiar with the concept of a computation graph, we will create one for the following function:

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

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

```
[25]: x = torch.arange(3, dtype=torch.float32, requires_grad=True) # Only float tensors can have gradients
print("X", x)
X tensor([0., 1., 2.], requires_grad=True)
```

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

```
[26]:
a = x + 2
b = a ** 2
c = b + 3
y = c.mean()
print("Y", y)
Y tensor(12.6667, grad_fn=<MeanBackward0>)
```

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

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

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

\( x \).\( \text{grad} \) will now contain the gradient \( \frac{\partial y}{\partial x} \), and this gradient indicates how a change in \( x \) will affect output \( y \) given the current input \( x = [0, 1, 2] \):

```
[28]: print(x.grad)
tensor([1.3333, 2.0000, 2.6667])
```

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

\[
\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial c_i} \frac{\partial c_i}{\partial b_i} \frac{\partial b_i}{\partial a_i} \frac{\partial a_i}{\partial x_i}
\]

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

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

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

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

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central Processing Unit</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>Several cores</td>
<td>Many cores</td>
</tr>
<tr>
<td>Low latency</td>
<td>High throughput</td>
</tr>
<tr>
<td>Good for serial processing</td>
<td>Good for parallel processing</td>
</tr>
<tr>
<td>Can do a handful of operations at once</td>
<td>Can do thousands of operations at once</td>
</tr>
</tbody>
</table>

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

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

```python
[29]: gpu_avail = torch.cuda.is_available()
print("Is the GPU available? %s \n%str(gpu_avail))
Is the GPU available? True
```

If you have a GPU on your computer but the command above returns False, make sure you have the correct CUDA-version installed. The d12020 environment comes with the CUDA-toolkit 10.1, which is selected for the Lisa supercomputer. Please change it if necessary (CUDA 10.2 is currently common). On Google Colab, make sure that you have selected a GPU in your runtime setup (in the menu, check under Runtime -> Change runtime type).

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

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

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

```python
[31]: x = torch.zeros(2, 3)
x = x.to(device)
print("X", x)
X tensor([[0., 0., 0.],
          [0., 0., 0.]], device='cuda:0')
```
In case you have a GPU, you should now see the attribute `device='cuda:0'` being printed next to your tensor. The zero next to cuda indicates that this is the zero-th GPU device on your computer. PyTorch also supports multi-GPU systems, but this you will only need once you have very big networks to train (if interested, see the PyTorch documentation). We can also compare the runtime of a large matrix multiplication on the CPU with an operation on the GPU:

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

    ## CPU version
    start_time = time.time()
    _ = torch.matmul(x, x)
    end_time = time.time()
    print("CPU time: %6.5f s" % (end_time - start_time))

    ## GPU version
    x = x.to(device)
    # The first operation on a CUDA device can be slow as it has to establish a CPU-GPU communication first.
    # Hence, we run an arbitrary command first without timing it for a fair comparison.
    if torch.cuda.is_available():
        _ = torch.matmul(x*0.0, x)
    start_time = time.time()
    _ = torch.matmul(x, x)
    end_time = time.time()
    print("GPU time: %6.5f s" % (end_time - start_time))

CPU time: 0.19239s
GPU time: 0.00008s
```

Depending on the size of the operation and the CPU/GPU in your system, the speedup of this operation can be >500x. As `matmul` operations are very common in neural networks, we can already see the great benefit of training a NN on a GPU. The time estimate can be relatively noisy here because we haven’t run it for multiple times. Feel free to extend this, but it also takes longer to run.

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

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

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

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

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

The model

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

```python
import torch.nn as nn
```

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

```python
import torch.nn.functional as F
```

nn.Module

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

```python
class MyModule(nn.Module):
    def __init__(self):
        super().__init__()
        # Some init for my module

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

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

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

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

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

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

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

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

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

```python
for name, param in model.named_parameters():
    print("Parameter \$s, shape \$s" % (name, str(param.shape)))
```

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

**The data**

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

```python
import torch.utils.data as data
```

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

**The dataset class**

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

```python
class XORDataset(data.Dataset):
    def __init__(self, size, std=0.1):
        """
        Inputs:
        size - Number of data points we want to generate
        std - Standard deviation of the noise (see generate_continuous_xor)
        """
        super().__init__()
        self.size = size
        self.std = std
        self.generate_continuous_xor()

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

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

(continues on next page)
return self.size

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

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

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

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

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

```python
def visualize_samples(data, label):
    if isinstance(data, torch.Tensor):
        data = data.cpu().numpy()
    if isinstance(label, torch.Tensor):
        label = label.cpu().numpy()
    data_0 = data[label == 0]
    data_1 = data[label == 1]

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

visualize_samples(dataset.data, dataset.label)
plt.show()
```

40 Chapter 5. Current progress
The data loader class

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

- **batch_size**: Number of samples to stack per batch
- **shuffle**: If True, the data is returned in a random order. This is important during training for introducing stochasticity.
- **num_workers**: Number of subprocesses to use for data loading. The default, 0, means that the data will be loaded in the main process which can slow down training for datasets where loading a data point takes a considerable amount of time (e.g. large images). More workers are recommended for those, but can cause issues on Windows computers. For tiny datasets as ours, 0 workers are usually faster.
- **pin_memory**: If True, the data loader will copy Tensors into CUDA pinned memory before returning them. This can save some time for large data points on GPUs. Usually a good practice to use for a training set, but not necessarily for validation and test to save memory on the GPU.
- **drop_last**: If True, the last batch is dropped in case it is smaller than the specified batch size. This occurs when the dataset size is not a multiple of the batch size. Only potentially helpful during training to keep a consistent batch size.

Let’s create a simple data loader below:
```python
# next(iter(...)) catches the first batch of the data loader
# If shuffle is True, this will return a different batch every time we run this cell
# For iterating over the whole dataset, we can simple use "for batch in data_loader: ...
...

data_loader = data.DataLoader(dataset, batch_size=8, shuffle=True)

next(iter(data_loader))

# The shape of the outputs are [batch_size, d_1,...,d_N] where d_1,...,d_N are the
# dimensions of the data point returned from the dataset class
```

```
print("Data inputs", data_inputs.shape, 
        "\n", data_inputs)

print("Data labels", data_labels.shape, "\n", data_labels)
```

```text
Data inputs torch.Size([8, 2])
tensor([[ 0.9717, 0.9721],
        [-0.0062, 0.0062],
        [ 1.0661, 0.9880],
        [ 0.0368, 0.9528],
        [ 0.0765, 0.9931],
        [ 0.0678, 1.0342]])
```

```
Data labels torch.Size([8])
tensor([0, 1, 0, 1, 0, 0, 1, 1])
```

**Optimization**

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

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

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

**Loss modules**

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

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

where \(y\) are our labels, and \(x\) our predictions, both in the range of \([0, 1]\). However, PyTorch already provides a list of predefined loss functions which we can use (see here for a full list). For instance, for BCE, PyTorch has two modules: `nn.BCELoss()`, `nn.BCEWithLogitsLoss()`. While `nn.BCELoss` expects the inputs \(x\) to be in the range \([0, 1]\), i.e. the output of a sigmoid, `nn.BCEWithLogitsLoss` combines a sigmoid layer and the BCE loss in a single class. This version is numerically more stable than using a plain Sigmoid followed by a BCE loss because of the logarithms applied in the loss function. Hence, it is advised to use loss functions applied on “logits” where possible
For our model defined above, we therefore use the module `nn.BCEWithLogitsLoss`.

```python
loss_module = nn.BCEWithLogitsLoss()
```

### Stochastic Gradient Descent

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

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

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

### Training

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

```python
train_dataset = XORDataset(size=1000)
train_data_loader = data.DataLoader(train_dataset, batch_size=128, shuffle=True)
```

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

```python
# Push model to device. Has to be only done once
model.to(device)
```

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

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

---

5.4. Tutorial 2: Introduction to PyTorch

43
def train_model(model, optimizer, data_loader, loss_module, num_epochs=100):
    # Set model to train mode
    model.train()
    
    # Training loop
    for epoch in tqdm(range(num_epochs)):
        for data_inputs, data_labels in data_loader:
            # Step 1: Move input data to device (only strictly necessary if we use GPU)
            data_inputs, data_labels = data_inputs.to(device), data_labels.to(device)
            
            # Step 2: Run the model on the input data
            preds = model(data_inputs)
            preds = preds.squeeze(dim=1)  # Output is [Batch size, 1], but we want [Batch size]

            # Step 3: Calculate the loss
            loss = loss_module(preds, data_labels.float())
            
            # Step 4: Perform backpropagation
            # Before calculating the gradients, we need to ensure that they are all zero.
            optimizer.zero_grad()
            # Perform backpropagation
            loss.backward()
            
            # Step 5: Update the parameters
            optimizer.step()

train_model(model, optimizer, train_data_loader, loss_module)

Saving a model

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

```
state_dict = model.state_dict()
print(state_dict)

OrderedDict([('linear1.weight', tensor([-2.0565, -2.2963, [ 1.2971, -1.8472],
                                                [-1.5020, -0.4974], [-0.6973, -0.8355]], device='cuda:0')),
             ('linear1.bias', tensor([ 0.7125, -0.8742, 1.2787, -0.2221], device='cuda:0')),
             ('linear2.weight', tensor([-2.5847, 1.9542, 2.1130, -0.3597], device='cuda:0')),
             ('linear2.bias', tensor([-0.9603, device='cuda:0']))])
```

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

```python
# Load state dict from the disk (make sure it is the same name as above)
state_dict = torch.load("our_model.tar")
# Create a new model and load the state
new_model = SimpleClassifier(num_inputs=2, num_hidden=4, num_outputs=1)
new_model.load_state_dict(state_dict)
# Verify that the parameters are the same
print("Original model\n", model.state_dict())
print("\nLoaded model\n", new_model.state_dict())
```

Original model

```
OrderedDict([('linear1.weight', tensor([[-2.0565, -2.2963],
  [ 1.2971, -1.8472],
  [-1.5020, -0.4974],
  [-0.6973, -0.8355]], device='cuda:0')), ('linear1.bias', tensor([ 0.7125, -0.8742, 1.2787, -0.2221]], device='cuda:0'))
```

Loaded model

```
OrderedDict([('linear1.weight', tensor([[-2.0565, -2.2963],
  [ 1.2971, -1.8472],
  [-1.5020, -0.4974],
  [-0.6973, -0.8355]])), ('linear1.bias', tensor([ 0.7125, -0.8742, 1.2787, -0.2221])), ('linear2.weight', tensor([[-2.5847, 1.9542, 2.1130, -0.3597]])), ('linear2.bias', tensor([-0.9603]], device='cuda:0'))
```

A detailed tutorial on saving and loading models in PyTorch can be found here.

**Evaluation**

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

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

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

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

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

When evaluating the model, we don’t need to keep track of the computation graph as we don’t intend to calculate the gradients. This reduces the required memory and speed up the model. In PyTorch, we can deactivate the computation graph using `with torch.no_grad(): ....` Remember to additionally set the model to eval mode.
def eval_model(model, data_loader):
    model.eval()  # Set model to eval mode
    true_preds, num_preds = 0., 0.

    with torch.no_grad():  # Deactivate gradients for the following code
        for data_inputs, data_labels in data_loader:
            # Determine prediction of model on dev set
            data_inputs, data_labels = data_inputs.to(device), data_labels.to(device)
            preds = model(data_inputs)
            preds = preds.squeeze(dim=1)
            preds = torch.sigmoid(preds)  # Sigmoid to map predictions between 0 and 1
            pred_labels = (preds >= 0.5).long()  # Binarize predictions to 0 and 1

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

    acc = true_preds / num_preds
    print("Accuracy of the model: \$%.2f\% \" (100.0*acc)"

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

Visualizing classification boundaries

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

@torch.no_grad()  # Decorator, same effect as "with torch.no_grad(): ..." over the whole function.
def visualize_classification(model, data, label):
    if isinstance(data, torch.Tensor):
        data = data.cpu().numpy()
    if isinstance(label, torch.Tensor):
        label = label.cpu().numpy()
    data_0 = data[label == 0]
    data_1 = data[label == 1]
    plt.figure(figsize=(4,4))
    plt.scatter(data_0[:,0], data_0[:,1], edgecolor="#333", label="Class 0")
    plt.scatter(data_1[:,0], data_1[:,1], edgecolor="#333", label="Class 1")
    plt.title("Dataset samples")
    plt.ylabel(r"$x_2$")
    plt.xlabel(r"$x_1$")

(continues on next page)
plt.legend()

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

The decision boundaries might not look exactly as in the figure in the preamble of this section which can be caused by running it on CPU or a different GPU architecture. Nevertheless, the result on the accuracy metric should be the approximately the same.
5.4.3 Additional features we didn’t get to discuss yet

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

5.5 Tutorial 3: Activation Functions

Filled notebook:

Pre-trained models:

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

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

```python
## Standard libraries
import os
import json
import math
import numpy as np

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

## Progress bar
from tqdm.notebook import tqdm

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

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

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

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

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

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

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

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

```python
[3]: import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial3/"
# Files to download
pretrained_files = ["FashionMNIST_elu.config", "FashionMNIST_elu.tar",
                   "FashionMNIST_leakyrelu.config", "FashionMNIST_leakyrelu.tar",
                   "FashionMNIST_relu.config", "FashionMNIST_relu.tar",
                   "FashionMNIST_sigmoid.config", "FashionMNIST_sigmoid.tar",
                   "FashionMNIST_swish.config", "FashionMNIST_swish.tar",
                   "FashionMNIST_tanh.config", "FashionMNIST_tanh.tar"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print("Downloading $s..." % file_url)
        try:
            urllib.request.urlretrieve(file_url, file_path)
            print("Downloaded: $s" % file_url)
        except HTTPError as e:
            print(e)
            print("Failed to download: $s due to an error.
```
5.5.1 Common activation functions

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

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

```python
[4]: class ActivationFunction(nn.Module):
    def __init__(self):
        super().__init__()
        self.name = self.__class__.__name__
        self.config = {"name": self.name}
```

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

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

```python
[5]:
    class Sigmoid(ActivationFunction):
        def forward(self, x):
            return 1 / (1 + torch.exp(-x))

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

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

```python
# Let's implement the four activation functions below:

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

class LeakyReLU(ActivationFunction):
    def __init__(self, alpha=0.1):
        super().__init__()
        self.config['alpha'] = alpha

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

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

class Swish(ActivationFunction):
    def forward(self, x):
        return x * torch.sigmoid(x)
```

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

```python
# Act_fn_by_name
act_fn_by_name = {
    "sigmoid": Sigmoid,
    "tanh": Tanh,
    "relu": ReLU,
    "leakyrelu": LeakyReLU,
    "elu": ELU,
    "swish": Swish
}
```
Visualizing activation functions

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

```python
def get_grads(act_fn, x):
    """Computes the gradients of an activation function at specified positions.
    Inputs:
    act_fn - An object of the class "ActivationFunction" with an implemented
    → forward pass.
    x - 1D input tensor.
    Output:
    A tensor with the same size of x containing the gradients of act_fn at x.
    """
    x = x.clone().requires_grad_()  # Mark the input as tensor for which we want to
    → store gradients
    out = act_fn(x)
    out.sum().backward()  # Summing results in an equal gradient flow to each element
    in x
    return x.grad  # Accessing the gradients of x by "x.grad"
```

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

```python
def vis_act_fn(act_fn, ax, x):
    # Run activation function
    y = act_fn(x)
    y_grads = get_grads(act_fn, x)
    # Push x, y and gradients back to cpu for plotting
    x, y, y_grads = x.cpu().numpy(), y.cpu().numpy(), y_grads.cpu().numpy()
    ## Plotting
    ax.plot(x, y, linewidth=2, label="ActFn")
    ax.plot(x, y_grads, linewidth=2, label="Gradient")
    ax.set_title(act_fn.name)
    ax.legend()
    ax.set_ylim(-1.5, x.max())
    # Add activation functions if wanted
    act_fns = [act_fn() for act_fn in act_fn_by_name.values()]
    x = torch.linspace(-5, 5, 1000)  # Range on which we want to visualize the activation
    → functions
    ## Plotting
    rows = math.ceil(len(act_fns)/2.0)
    fig, ax = plt.subplots(rows, 2, figsize=(8, rows*4))
    for i, act_fn in enumerate(act_fns):
        vis_act_fn(act_fn, ax[divmod(i,2)], x)
    fig.subplots_adjust(hspace=0.3)
    plt.show()
```
5.5. Tutorial 3: Activation Functions

- Sigmoid
- Tanh
- ReLU
- LeakyReLU
- ELU
- Swish
5.5.2 Analysing the effect of activation functions

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

Setup

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

```python
class BaseNetwork(nn.Module):
    def __init__(self, act_fn, input_size=784, num_classes=10, hidden_sizes=[512, 256, 256, 128]):
        """
        Inputs:
        act_fn - Object of the activation function that should be used as non-linearity in the network.
        input_size - Size of the input images in pixels
        num_classes - Number of classes we want to predict
        hidden_sizes - A list of integers specifying the hidden layer sizes in the NN
        """
        super().__init__()
        # Create the network based on the specified hidden sizes
        layers = []
        layer_sizes = [input_size] + hidden_sizes
        for layer_index in range(1, len(layer_sizes)):
            layers += [nn.Linear(layer_sizes[layer_index-1], layer_sizes[layer_index]), act_fn]
        layers += [nn.Linear(layer_sizes[-1], num_classes)]
        self.layers = nn.Sequential(*layers)
        # nn.Sequential summarizes a list of modules into a single module, applying them in sequence
        # We store all hyperparameters in a dictionary for saving and loading of the model
        self.config = {"act_fn": act_fn.config, "input_size": input_size, "num_classes": num_classes, "hidden_sizes": hidden_sizes}
        def forward(self, x):
            x = x.view(x.size(0), -1) # Reshape images to a flat vector
            out = self.layers(x)
            return out
```

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

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

def _get_model_file(model_path, model_name):
    # Name of the file for storing network parameters
```

(continues on next page)
return os.path.join(model_path, model_name + ".tar")

def load_model(model_path, model_name, net=None):
    """
    Loads a saved model from disk.
    Inputs:
    model_path - Path of the checkpoint directory
    model_name - Name of the model (str)
    net - (Optional) If given, the state dict is loaded into this model.
    Otherwise, a new model is created.
    """
    config_file, model_file = _get_config_file(model_path, model_name), _get_model_file(model_path, model_name)
    assert os.path.isfile(config_file), "Could not find the config file "%s". Are you sure this is the correct path and you have your model config stored here?" % (config_file)
    assert os.path.isfile(model_file), "Could not find the model file "%s". Are you sure this is the correct path and you have your model stored here?" % (model_file)
    with open(config_file, "r") as f:
        config_dict = json.load(f)
    if net is None:
        act_fn_name = config_dict["act_fn"]["name"].lower()
        act_fn = act_fn_by_name[act_fn_name](**config_dict.pop("act_fn"))
        net = BaseNetwork(act_fn=act_fn, **config_dict)
    net.load_state_dict(torch.load(model_file, map_location=device))
    return net

def save_model(model, model_path, model_name):
    """
    Given a model, we save the state dict and hyperparameters.
    Inputs:
    model - Network object to save parameters from
    model_path - Path of the checkpoint directory
    model_name - Name of the model (str)
    """
    config_dict = model.config
    os.makedirs(model_path, exist_ok=True)
    config_file, model_file = _get_config_file(model_path, model_name), _get_model_file(model_path, model_name)
    with open(config_file, "w") as f:
        json.dump(config_dict, f)
    torch.save(model.state_dict(), model_file)

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

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

[12]: import torchvision
from torchvision.datasets import FashionMNIST
(continues on next page)
from torchvision import transforms

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

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

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

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

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

plt.figure(figsize=(8,8))
plt.title("FashionMNIST examples")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
Visualizing the gradient flow after initialization

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

To get a feeling of how every activation function influences the gradients, we can look at a freshly initialized network and measure the gradients for each parameter for a batch of 256 images:
```python
[14]: def visualize_gradients(net, color="C0"):
    
    Inputs:
    net - Object of class BaseNetwork
    color - Color in which we want to visualize the histogram (for easier
    separation of activation functions)
    
    net.eval()
    small_loader = data.DataLoader(train_set, batch_size=256, shuffle=False)
    imgs, labels = next(iter(small_loader))
    imgs, labels = imgs.to(device), labels.to(device)
    
    # Pass one batch through the network, and calculate the gradients for the weights
    net.zero_grad()
    preds = net(imgs)
    loss = F.cross_entropy(preds, labels)
    loss.backward()
    
    # We limit our visualization to the weight parameters and exclude the bias to
    # reduce the number of plots
    grads = {name: params.grad.data.view(-1).cpu().clone().numpy() for name, params
    in net.named_parameters() if "weight" in name}
    net.zero_grad()
    
    ## Plotting
    columns = len(grads)
    fig, ax = plt.subplots(1, columns, figsize=(columns*3.5, 2.5))
    fig_index = 0
    for key in grads:
        key_ax = ax[fig_index%columns]
        sns.histplot(data=grads[key], bins=30, ax=key_ax, color=color, kde=True)
        key_ax.set_title("%s" % key)
        key_ax.set_xlabel("Grad magnitude")
        fig_index += 1
    fig.suptitle("Gradient magnitude distribution for activation function %s" % (net.
    config["act_fn"]['name']), fontsize=14, y=1.05)
    fig.subplots_adjust(wspace=0.45)
    plt.show()
    plt.close()

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

---

**Chapter 5. Current progress**

---
The sigmoid activation function shows a clearly undesirable behavior. While the gradients for the output layer are very large with up to 0.1, the input layer has the lowest gradient norm across all activation functions with only 1e-5. This
is due to its small maximum gradient of 1/4, and finding a suitable learning rate across all layers is not possible in this setup. All the other activation functions show to have similar gradient norms across all layers. Interestingly, the ReLU activation has a spike around 0 which is caused by its zero-part on the left, and dead neurons (we will take a closer look at this later on).

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

### Training a model

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

```python
def train_model(net, model_name, max_epochs=50, patience=7, batch_size=256, overwrite=False):
    """
    Train a model on the training set of FashionMNIST
    Inputs:
    net - Object of BaseNetwork
    model_name - (str) Name of the model, used for creating the checkpoint names
    max_epochs - Number of epochs we want to (maximally) train for
    patience - If the performance on the validation set has not improved for
               # patience epochs, we stop training early
    batch_size - Size of batches used in training
    overwrite - Determines how to handle the case when there already exists a
                checkpoint. If True, it will be overwritten. Otherwise, we skip training.
    """
    file_exists = os.path.isfile(_get_model_file(CHECKPOINT_PATH, model_name))
    if file_exists and not overwrite:
        print("Model file already exists. Skipping training...")
    else:
        if file_exists:
            print("Model file exists, but will be overwritten...")

        # Defining optimizer, loss and data loader
        optimizer = optim.SGD(net.parameters(), lr=1e-2, momentum=0.9) # Default
        loss_module = nn.CrossEntropyLoss()
        train_loader_local = data.DataLoader(train_set, batch_size=batch_size, shuffle=True, drop_last=True, pin_memory=True)
        val_scores = []
        best_val_epoch = -1
        for epoch in range(max_epochs):
            ##############
            ## Training ##
            ##############
            net.train()
            true_preds, count = 0., 0
            for imgs, labels in tqdm(train_loader, desc="Epoch %i" % (epoch+1), leave=False):
                imgs, labels = imgs.to(device), labels.to(device) # To GPU
                optimizer.zero_grad() # Zero-grad can be placed anywhere before "loss. backward()"
                (continues on next page)"""
```
preds = net(imgs)
loss = loss_module(preds, labels)
loss.backward()
# Record statistics during training
true_preds += (preds.argmax(dim=-1) == labels).sum()
count += labels.shape[0]
train_acc = true_preds / count

################
## Validation ##
################
val_acc = test_model(net, val_loader)
val_scores.append(val_acc)
print('[Epoch %2i] Training accuracy: %05.2f%%, Validation accuracy: %05.2f%%' % (epoch+1, train_acc*100.0, val_acc*100.0))
if len(val_scores) == 1 or val_acc > val_scores[best_val_epoch]:
    print('  (New best performance, saving model...)
    save_model(net, CHECKPOINT_PATH, model_name)
    best_val_epoch = epoch
elif best_val_epoch <= epoch - patience:
    print('Early stopping due to no improvement over the last %i epochs' % (patience))
    break

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

load_model(CHECKPOINT_PATH, model_name, net=net)
test_acc = test_model(net, test_loader)
print(" Test accuracy: %4.2f%%")
return test_acc

def test_model(net, data_loader):
    ""
    Test a model on a specified dataset.
    ""
    net.eval()
    true_preds, count = 0., 0
    for imgs, labels in data_loader:
        imgs, labels = imgs.to(device), labels.to(device)
        with torch.no_grad():
            preds = net(imgs).argmax(dim=-1)
            true_preds += (preds == labels).sum().item()
            count += labels.shape[0]
        test_acc = true_preds / count
        test_acc = test_acc
(continues on next page)
We train one model for each activation function. We recommend using the pretrained models to save time if you are running this notebook on CPU.

```python
[17]: for act_fn_name in act_fn_by_name:
    print("Training BaseNetwork with \$s activation..." \ % act_fn_name)
    set_seed(42)
    act_fn = act_fn_by_name[act_fn_name]()
    net_actfn = BaseNetwork(act_fn=act_fn).to(device)
    train_model(net_actfn, "FashionMNIST_\$s" \ % act_fn_name, overwrite=False)
```

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

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

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

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

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

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

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

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

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

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

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

    ## Plotting
    columns = 4
    rows = math.ceil(len(activations)/columns)
    fig, ax = plt.subplots(rows, columns, figsize=(columns*2.7, rows*2.5))
    fig_index = 0
    for key in activations:
        key_ax = ax[fig_index//columns][fig_index%columns]
        sns.histplot(data=activations[key], bins=50, ax=key_ax, color=color, kde=True, stat="density")
        key_ax.set_title("Layer %i - %s" % (key, net.layers[key].__class__.__name__))
        fig_index += 1

    fig.suptitle("Activation distribution for activation function %s" % (net.config["act_fn"])["name"], fontsize=14)
    fig.subplots_adjust(hspace=0.4, wspace=0.4)
    plt.show()
    plt.close()

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

5.5. Tutorial 3: Activation Functions 63
Activation distribution for activation function Sigmoid

Activation distribution for activation function Tanh
5.5. Tutorial 3: Activation Functions
As the model with sigmoid activation was not able to train properly, the activations are also less informative and all gathered around 0.5 (the activation at input 0).

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

The ReLU has a strong peak at 0, as we initially expected. The effect of having no gradients for negative values is that the network does not have a Gaussian-like distribution after the linear layers, but a longer tail towards the positive values. The LeakyReLU shows a very similar behavior while ELU follows again a more Gaussian-like distribution.
The Swish activation seems to lie in between, although it is worth noting that Swish uses significantly higher values than other activation functions (up to 20).

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

Finding dead neurons in ReLU networks

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### 5.5.3 Conclusion

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


5.6 Tutorial 4: Optimization and Initialization

Filled notebook:

Pre-trained models:

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

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

Let’s start with importing our standard libraries:

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

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

    ## Progress bar
    from tqdm.notebook import tqdm

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

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

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

    # Path to the folder where the pretrained models are saved
    CHECKPOINT_PATH = "../saved_models/tutorial4"
```

# Function for setting the seed

def set_seed(seed):
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)

set_seed(42)

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

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

Using device cuda:0

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

```
[3]:

import urllib.request
from urllib.error import HTTPError

# Github URL where saved models are stored for this tutorial
# Files to download
pretrained_files = ["FashionMNIST_SGD.config", "FashionMNIST_SGD_results.json",
                                      "FashionMNIST_SGD.tar",
                                      "FashionMNIST_SGDMom.config", "FashionMNIST_SGDMom_results.json",
                                      "FashionMNIST_SGDMom.tar",
                                      "FashionMNIST_Adam.config", "FashionMNIST_Adam_results.json",
                                      "FashionMNIST_Adam.tar"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

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

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

```python
from torchvision.datasets import FashionMNIST
from torchvision import transforms

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

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

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

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

In comparison to the previous tutorial, we have changed the parameters of the normalization transformation `transforms.Normalize`. The normalization is now designed to give us an expected mean of 0 and a standard deviation of 1 across pixels. This will be particularly relevant for the discussion about initialization we will look at below, and hence we change it here. It should be noted that in most classification tasks, both normalization techniques (between -1 and 1 or mean 0 and stddev 1) have shown to work well. We can calculate the normalization parameters by determining the mean and standard deviation on the original images:

```python
print("Mean", (train_dataset.data.float() / 255.0).mean().item())
print("Std", (train_dataset.data.float() / 255.0).std().item())
Mean 0.2860923707485199
Std 0.3530242443084717
```

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

```python
imgs, _ = next(iter(train_loader))
print("Mean: $5.3f" % imgs.mean().item())
print("Standard deviation: $5.3f" % imgs.std().item())
print("Maximum: $5.3f" % imgs.max().item())
print("Minimum: $5.3f" % imgs.min().item())
Mean: 0.002
Standard deviation: 1.001
Maximum: 2.022
```

(continues on next page)
Minimum: -0.810

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

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

```python
class BaseNetwork(nn.Module):
    def __init__(self, act_fn, input_size=784, num_classes=10, hidden_sizes=[512, 256, 256, 128]):
        """
        Inputs:
        act_fn - Object of the activation function that should be used as non-linearity in the network.
        input_size - Size of the input images in pixels
        num_classes - Number of classes we want to predict
        hidden_sizes - A list of integers specifying the hidden layer sizes in the NN
        """
        super().__init__()
        layers = []
        layer_sizes = [input_size] + hidden_sizes
        for layer_index in range(1, len(layer_sizes)):
            layers += [nn.Linear(layer_sizes[layer_index-1], layer_sizes[layer_index]), act_fn]
        layers += [nn.Linear(layer_sizes[-1], num_classes)]
        self.layers = nn.ModuleList(layers)  # A module list registers a list of modules as submodules (e.g. for parameters)
        self.config = {"act_fn": act_fn.__class__.__name__, "input_size": input_size, "num_classes": num_classes, "hidden_sizes": hidden_sizes}

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

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

```python
class Identity(nn.Module):
    def forward(self, x):
        return x
```

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

[9]: #*******************************************************************************

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

def visualize_weight_distribution(model, color="C0"):
    weights = {}
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            continue
        key_name = "Layer %s" % name.split(".")[1]
        weights[key_name] = param.detach().view(-1).cpu().numpy()

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

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

    # Pass one batch through the network, and calculate the gradients for the weights
    model.zero_grad()
    preds = model(imgs)
    loss = F.cross_entropy(preds, labels) # Same as nn.CrossEntropyLoss, but as a
    # function instead of module

    (continues on next page)
loss.backward()

# We limit our visualization to the weight parameters and exclude the bias to
# reduce the number of plots
grads = {name: params.grad.view(-1).cpu().clone().numpy() for name, params in model.named_parameters() if "weight" in name}
model.zero_grad()

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

if print_variance:
    for key in sorted(grads.keys()):
        print("%s - Variance: %f" % (key, np.var(grads[key])))

##############################################################

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

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

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

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

##############################################################
### 5.6.2 Initialization

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

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

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

```python
model = BaseNetwork(act_fn=Identity()).to(device)
```

#### Constant initialization

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

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

const_init(model, c=0.005)
visualize_gradients(model)
visualize_activations(model, print_variance=True)
```
As we can see, only the first and the last layer have diverse gradient distributions while the other three layers have the same gradient for all weights (note that this value is unequal 0, but often very close to it). Having the same gradient for parameters that have been initialized with the same values means that we will always have the same value for those parameters. This would make our layer useless and reduce our effective number of parameters to 1. Thus, we cannot use a constant initialization to train our networks.

**Constant variance**

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

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

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

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

```python
[13]: var_init(model, std=0.1)
visualize_activations(model, print_variance=True)
```
<table>
<thead>
<tr>
<th>Layer</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7.901733</td>
</tr>
<tr>
<td>2</td>
<td>38.941471</td>
</tr>
<tr>
<td>4</td>
<td>103.246284</td>
</tr>
<tr>
<td>6</td>
<td>262.478546</td>
</tr>
<tr>
<td>8</td>
<td>248.354202</td>
</tr>
</tbody>
</table>

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

**How to find appropriate initialization values**

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

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

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

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

$$ y_i = \sum_j w_{ij} x_j \quad \text{Calculation of a single output neuron without bias}$$

$$ \text{Var}(y_i) = \sigma_x^2 = \text{Var} \left( \sum_j w_{ij} x_j \right) $$

$$ = \sum_j \text{Var}(w_{ij} x_j) \quad \text{Inputs and weights are independent of each other}$$

$$ = \sum_j \text{Var}(w_{ij}) \cdot \text{Var}(x_j) \quad \text{Variance rule (see above) with expectations being zero}$$

$$ = d_x \cdot \text{Var}(w_{ij}) \cdot \text{Var}(x_j) \quad \text{Variance equal for all } d_x \text{ elements}$$

$$ = \sigma_x^2 \cdot d_x \cdot \text{Var}(w_{ij})$$

$$ \Rightarrow \text{Var}(w_{ij}) = \sigma_w^2 = \frac{1}{d_x}$$

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

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

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

Layer 0 - Variance: 1.020319
Layer 2 - Variance: 1.049295
Layer 4 - Variance: 1.031418
Layer 6 - Variance: 1.025792
Layer 8 - Variance: 0.872356
As we expected, the variance stays indeed constant across layers. Note that our initialization does not restrict us to a normal distribution, but allows any other distribution with a mean of 0 and variance of \( \frac{1}{d_x} \). You often see that a uniform distribution is used for initialization. A small benefit of using a uniform instead of a normal distribution is that we can exclude the chance of initializing very large or small weights.

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

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

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

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

Let’s shortly implement it and validate its effectiveness:

```python
[15]: def xavier_init(model):
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            param.data.fill_(0)
        else:
            bound = math.sqrt(6)/math.sqrt(param.shape[0]+param.shape[1])
            param.data.uniform_(-bound, bound)

xavier_init(model)
visualize_gradients(model, print_variance=True)
visualize_activations(model, print_variance=True)
```

<table>
<thead>
<tr>
<th>layers.0.weight</th>
<th>layers.2.weight</th>
<th>Gradient distribution</th>
<th>layers.4.weight</th>
<th>layers.6.weight</th>
<th>layers.8.weight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grad magnitude</td>
<td>Grad magnitude</td>
<td>Grad magnitude</td>
<td>Grad magnitude</td>
<td>Grad magnitude</td>
<td>Grad magnitude</td>
</tr>
<tr>
<td>Count</td>
<td>Count</td>
<td>Count</td>
<td>Count</td>
<td>Count</td>
<td>Count</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10000</td>
<td>25000</td>
<td>12500</td>
<td>3000</td>
<td>5000</td>
<td>12500</td>
</tr>
<tr>
<td>20000</td>
<td>25000</td>
<td>12500</td>
<td>3000</td>
<td>5000</td>
<td>12500</td>
</tr>
<tr>
<td>30000</td>
<td>25000</td>
<td>12500</td>
<td>3000</td>
<td>5000</td>
<td>12500</td>
</tr>
</tbody>
</table>

layers.0.weight - Variance: 0.000436
layers.2.weight - Variance: 0.000747
layers.4.weight - Variance: 0.001149
layers.6.weight - Variance: 0.001744
layers.8.weight - Variance: 0.017655
Layer 0 - Variance: 1.216592
Layer 2 - Variance: 1.719161
Layer 4 - Variance: 1.714506
Layer 6 - Variance: 2.224779
Layer 8 - Variance: 5.297660

We see that the Xavier initialization balances the variance of gradients and activations. Note that the significantly higher variance for the output layer is due to the large difference of input and output dimension (128 vs 10). However, we currently assumed the activation function to be linear. So what happens if we add a non-linearity? In a tanh-based network, a common assumption is that for small values during the initial steps in training, the tanh works as a linear function such that we don’t have to adjust our calculation. We can check if that is the case for us as well:

```
model = BaseNetwork(act_fn=nn.Tanh()).to(device)
xavier_init(model)
visualize_gradients(model, print_variance=True)
visualize_activations(model, print_variance=True)
```

Layer 0 - Variance: 1.295969
Layer 2 - Variance: 0.583388

(continues on next page)
Although the variance decreases over depth, it is apparent that the activation distribution becomes more focused on the low values. Therefore, our variance will stabilize around 0.25 if we would go even deeper. Hence, we can conclude that the Xavier initialization works well for Tanh networks. But what about ReLU networks? Here, we cannot take the previous assumption of the non-linearity becoming linear for small values. The ReLU activation function sets (in expectation) half of the inputs to 0 so that also the expectation of the input is not zero. However, as long as the expectation of $W$ is zero and $b = 0$, the expectation of the output is zero. The part where the calculation of the ReLU initialization differs from the identity is when determining $\text{Var}(w_{ij}x_j)$:

$$\text{Var}(w_{ij}x_j) = \frac{E[w_{ij}^2]}{\text{Var}(w_{ij})} E[x_j^2] - \frac{E[w_{ij}]^2}{0} E[x_j]^2 = \text{Var}(w_{ij})E[x_j^2]$$

If we assume now that $x$ is the output of a ReLU activation (from a previous layer, $x = \max(0, \tilde{y})$), we can calculate the expectation as follows:

$$E[x^2] = E[\max(0, \tilde{y})^2]$$

$$= \frac{1}{2} E[\tilde{y}^2]$$

$\tilde{y}$ is zero-centered and symmetric

$$= \frac{1}{2} \text{Var}(\tilde{y})$$

Thus, we see that we have an additional factor of $1/2$ in the equation, so that our desired weight variance becomes $2/d_x$. This gives us the Kaiming initialization (see He, K. et al. (2015)). Note that the Kaiming initialization does not use the harmonic mean between input and output size. In their paper (Section 2.2, Backward Propagation, last paragraph), they argue that using $d_x$ or $d_y$ both lead to stable gradients throughout the network, and only depend on the overall input and output size of the network. Hence, we can use here only the input $d_x$:

```python
[17]: def kaiming_init(model):
    for name, param in model.named_parameters():
        if name.endswith("bias"):  
            param.data.fill_(0)  
        elif name.startswith("layers.0"):  # The first layer does not have ReLU  
            applied on its input
            param.data.normal_(0, 1/math.sqrt(param.shape[1]))  
        else:
            param.data.normal_(0, math.sqrt(2)/math.sqrt(param.shape[1]))

model = BaseNetwork(act_fn=nn.ReLU()).to(device)
kaiming_init(model)
visualize_gradients(model, print_variance=True)
visualize_activations(model, print_variance=True)
```
The variance stays stable across layers. We can conclude that the Kaiming initialization indeed works well for ReLU-based networks. Note that for Leaky-ReLU etc., we have to slightly adjust the factor of 2 in the variance as half of the values are not set to zero anymore. PyTorch provides a function to calculate this factor for many activation function, see torch.nn.init.calculate_gain(link).

5.6.3 Optimization

Besides initialization, selecting a suitable optimization algorithm can be an important choice for deep neural networks. Before taking a closer look at them, we should define code for training the models. Most of the following code is copied from the previous tutorial, and only slightly altered to fit our needs.

```python
[18]: def _get_config_file(model_path, model_name):
    return os.path.join(model_path, model_name + ".config")

def _get_model_file(model_path, model_name):
    return os.path.join(model_path, model_name + ".tar")

def _get_result_file(model_path, model_name):
    return os.path.join(model_path, model_name + ".results.json")

def load_model(model_path, model_name, net=None):
    config_file, model_file = _get_config_file(model_path, model_name), _get_model_file(model_path, model_name)
    assert os.path.isfile(config_file), "Could not find the config file " + repr(config_file) + ".
    assert os.path.isfile(model_file), "Could not find the model file " + repr(model_file) + ".
    with open(config_file, "r") as f:
        config_dict = json.load(f)
    if net is None:
        act_fn_name = config_dict["act_fn_name"].pop("name").lower()
        assert act_fn_name in act_fn_by_name, "Unknown activation function " + repr(act_fn_name) + ". Please add it to the " + repr(act_fn_by_name) + ".dict."
act_fn = act_fn_by_name[act_fn_name]()
net = BaseNetwork(act_fn=act_fn, **config_dict)
net.load_state_dict(torch.load(model_file))
return net

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

def train_model(net, model_name, optim_func, max_epochs=50, batch_size=256, overwrite=False):
    """
    Train a model on the training set of FashionMNIST
    """
    file_exists = os.path.isfile(_get_model_file(CHECKPOINT_PATH, model_name))
    if file_exists and not overwrite:
        print("Model file of \"%s\" already exists. Skipping training..." % model_name)
    else:
        if file_exists:
            with open(_get_result_file(CHECKPOINT_PATH, model_name), "r") as f:
                results = json.load(f)
        else:
            print("Model file exists, but will be overwritten...")

            # Defining optimizer, loss and data loader
            optimizer = optim_func(net.parameters())
            loss_module = nn.CrossEntropyLoss()
            train_loader_local = data.DataLoader(train_set, batch_size=batch_size, shuffle=True, drop_last=True, pin_memory=True)

            results = None
            val_scores = []
            train_losses, train_scores = [], []
            best_val_epoch = -1
            for epoch in range(max_epochs):
                ##############
                ## Training ##
                ##############
                net.train()
                true_preds, count = 0., 0
                t = tqdm(train_loader_local, leave=False)
                for imgs, labels in t:
(continues on next page)
imgs, labels = imgs.to(device), labels.to(device)
 optimizer.zero_grad()
 preds = net(imgs)
 loss = loss_module(preds, labels)
 loss.backward()
 optimizer.step()

 # Record statistics during training
 true_preds += (preds.argmax(dim=-1) == labels).sum().item()
 count += labels.shape[0]
 t.set_description("Epoch %i: loss=%.2f" % (epoch+1, loss.item()))
 train_losses.append(loss.item())

 train_acc = true_preds / count
 train_scores.append(train_acc)

# Validation

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

 if len(val_scores) == 1 or val_acc > val_scores[best_val_epoch]:
   print("t(New best performance, saving model...)")
   save_model(net, CHECKPOINT_PATH, model_name)
   best_val_epoch = epoch

 if results is None:
   load_model(CHECKPOINT_PATH, model_name, net=net)
   test_acc = test_model(net, test_loader)
   results = {"test_acc": test_acc, "val_scores": val_scores, "train_losses": train_losses, "train_scores": train_scores}

 with open(_get_result_file(CHECKPOINT_PATH, model_name), "w") as f:
   json.dump(results, f)

 # Plot a curve of the validation accuracy
 sns.set()
 plt.plot([i for i in range(1,len(results["train_scores"])+1)], results["train_scores"], label="Train")
 plt.plot([i for i in range(1,len(results["val_scores"])+1)], results["val_scores"], label="Val")
 plt.xlabel("Epochs")
 plt.ylabel("Validation accuracy")
 plt.ylim(min(results["val_scores"]), max(results["train_scores"])*1.01)
 plt.title("Validation performance of %s" % model_name)
 plt.legend()
 plt.show()
 plt.close()

 print("Test accuracy: %.2f%% " % (results["test_acc"]*100.0)).center(50, "=")+"n")
 return results

def test_model(net, data_loader):
  ""
  Test a model on a specified dataset.
  
  (continues on next page)
Inputs:
- net - Trained model of type `BaseNetwork`
- data_loader - `DataLoader` object of the dataset to test on (validation or test)

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

First, we need to understand what an optimizer actually does. The optimizer is responsible to update the network’s parameters given the gradients. Hence, we effectively implement a function $w^t = f(w^{t-1}, g^t, ...)$ with $w$ being the parameters, and $g^t = \nabla_{w^{t-1}}\mathcal{L}^t$ the gradients at time step $t$. A common, additional parameter to this function is the learning rate, here denoted by $\eta$. Usually, the learning rate can be seen as the “step size” of the update. A higher learning rate means that we change the weights more in the direction of the gradients, a smaller means we take shorter steps.

As most optimizers only differ in the implementation of $f$, we can define a template for an optimizer in PyTorch below. We take as input the parameters of a model and a learning rate. The function `zero_grad` sets the gradients of all parameters to zero, which we have to do before calling `loss.backward()`. Finally, the `step()` function tells the optimizer to update all weights based on their gradients. The template is setup below:

```python
[19]: class OptimizerTemplate:

    def __init__(self, params, lr):
        self.params = list(params)
        self.lr = lr

    def zero_grad(self):
        # Set gradients of all parameters to zero
        for p in self.params:
            if p.grad is not None:
                p.grad.detach_() # For second-order optimizers important
                p.grad.zero_()

    @torch.no_grad()
    def step(self):
        # Apply update step to all parameters
        for p in self.params:
            if p.grad is None: # We skip parameters without any gradients
                continue
            self.update_param(p)

    def update_param(self, p):
        # To be implemented in optimizer-specific classes
        raise NotImplementedError
```

The first optimizer we are going to implement is the standard Stochastic Gradient Descent (SGD). SGD updates the parameters using the following equation:

$$w^{(t)} = w^{(t-1)} - \eta \cdot g^{(t)}$$

5.6. Tutorial 4: Optimization and Initialization 85
As simple as the equation is also our implementation of SGD:

```python
[20]: class SGD(OptimizerTemplate):
    def __init__(self, params, lr):
        super().__init__(params, lr)
    def update_param(self, p):
        p_update = -self.lr * p.grad
        p.add_(p_update)  # In-place update => saves memory and does not create computation graph

In the lecture, we also have discussed the concept of momentum which replaces the gradient in the update by an exponential average of all past gradients including the current one:

\[
\begin{align*}
    m(t) &= \beta_1 m(t-1) + (1 - \beta_1) \cdot g(t) \\
    w(t) &= w(t-1) - \eta \cdot m(t)
\end{align*}
\]

Let’s also implement it below:

```python
[21]: class SGDMomentum(OptimizerTemplate):
    def __init__(self, params, lr, momentum=0.0):
        super().__init__(params, lr)
        self.momentum = momentum  # Corresponds to beta_1 in the equation above
        self.param_momentum = {p: torch.zeros_like(p.data) for p in self.params}  # Dict to store m_t
    def update_param(self, p):
        p_update = -self.lr * self.param_momentum[p]
        p.add_(p_update)
```

Finally, we arrive at Adam. Adam combines the idea of momentum with an adaptive learning rate, which is based on an exponential average of the squared gradients, i.e. the gradients norm. Furthermore, we add a bias correction for the momentum and adaptive learning rate for the first iterations:

\[
\begin{align*}
    m(t) &= \beta_1 m(t-1) + (1 - \beta_1) \cdot g(t) \\
    v(t) &= \beta_2 v(t-1) + (1 - \beta_2) \cdot (g(t))^2 \\
    \hat{m}(t) &= m(t) / (1 - \beta_1^t), \hat{v}(t) = v(t) / (1 - \beta_2^t) \\
    w(t) &= w(t-1) - \eta \sqrt{\hat{v}(t) + \epsilon} \cdot \hat{m}(t)
\end{align*}
\]

Epsilon is a small constant used to improve numerical stability for very small gradient norms. Remember that the adaptive learning rate does not replace the learning rate hyperparameter \( \eta \), but rather acts as an extra factor and ensures that the gradients of various parameters have a similar norm.

```python
[22]: class Adam(OptimizerTemplate):
    def __init__(self, params, lr, beta1=0.9, beta2=0.999, eps=1e-8):
        super().__init__(params, lr)
        self.beta1 = beta1  # betal
        self.beta2 = beta2  # betal
```

(continues on next page)
Comparing optimizers on model training

After we have implemented three optimizers (SGD, SGD with momentum, and Adam), we can start to analyze and compare them. First, we test them on how well they can optimize a neural network on the FashionMNIST dataset. We use again our linear network, this time with a ReLU activation and the kaiming initialization, which we have found before to work well for ReLU-based networks. Note that the model is over-parameterized for this task, and we can achieve similar performance with a much smaller network (for example 100, 100, 100). However, our main interest is in how well the optimizer can train deep neural networks, hence the over-parameterization.

For a fair comparison, we train the exact same model with the same seed with the three optimizers below. Feel free to change the hyperparameters if you want (however, you have to train your own model then).

```python
[23]: base_model = BaseNetwork(act_fn=nn.ReLU(), hidden_sizes=[512,256,256,128])
kaiming_init(base_model)

Model file of "FashionMNIST_SGD" already exists. Skipping training...
```

5.6. Tutorial 4: Optimization and Initialization 87
Validation performance of FashionMNIST_SGD

Validation accuracy

Validation performance of FashionMNIST_SGD

Train
Val

Epochs

0 5 10 15 20 25 30 35 40

Validation accuracy

Epochs

0.84
0.86
0.88
0.90
0.92
0.94
0.96
0.98
1.00

Validation performance of FashionMNIST_SGD

Validation accuracy

Validation performance of FashionMNIST_SGD

Train
Val

Epochs

0 5 10 15 20 25 30 35 40

Validation accuracy

[25]: SGDMom_model = copy.deepcopy(base_model).to(device)
SGDMom_results = train_model(SGDMom_model, "FashionMNIST_SGDMom",
        lambda params: SGDMomentum(params, lr=1e-1, momentum=0.9),
        max_epochs=40, batch_size=256)
Model file of "FashionMNIST_SGDMom" already exists. Skipping training...

Chapter 5. Current progress
Validation performance of FashionMNIST_SGDMom

Validation accuracy

Epochs

[26]: Adam_model = copy.deepcopy(base_model).to(device)
Adam_results = train_model(Adam_model, "FashionMNIST_Adam",
                           lambda params: Adam(params, lr=1e-3),
                           max_epochs=40, batch_size=256)

Model file of "FashionMNIST_Adam" already exists. Skipping training...

================ Test accuracy: 88.83% =================
The result is that all optimizers perform similarly well with the given model. The differences are too small to find any significant conclusion. However, keep in mind that this can also be attributed to the initialization we chose. When changing the initialization to worse (e.g. constant initialization), Adam usually shows to be more robust because of its adaptive learning rate. To show the specific benefits of the optimizers, we will continue to look at some possible loss surfaces in which momentum and adaptive learning rate are crucial.

Pathological curvatures

A pathological curvature is a type of surface that is similar to ravines and is particularly tricky for plain SGD optimization. In words, pathological curvatures typically have a steep gradient in one direction with an optimum at the center, while in a second direction we have a slower gradient towards a (global) optimum. Let’s first create an example surface of this and visualize it:

```python
[27]: def pathological_curve_loss(w1, w2):
    # Example of a pathological curvature. There are many more possible, feel free to experiment here!
    x1_loss = torch.tanh(w1)**2 + 0.01 * torch.abs(w1)
    x2_loss = torch.sigmoid(w2)
    return x1_loss + x2_loss

[28]: def plot_curve(curve_fn, x_range=(-5,5), y_range=(-5,5), plot_3d=False, cmap=cm.viridis, title="Pathological curvature"):
    fig = plt.figure()
    ax = fig.gca(projection='3d')  #if plot_3d else fig.gca()
    x = torch.arange(x_range[0], x_range[1], (x_range[1]-x_range[0])/100.)
```

(continues on next page)
y = torch.arange(y_range[0], y_range[1], (y_range[1] - y_range[0]) / 100.)
x, y = torch.meshgrid([x, y])
z = curve_fn(x, y)
x, y, z = x.numpy(), y.numpy(), z.numpy()

if plot_3d:
    ax.plot_surface(x, y, z, cmap=cmap, linewidth=1, color="#000", antialiased=False)
    ax.set_zlabel("loss")
else:
    ax.imshow(z.T[::-1], cmap=cmap, extent=(x_range[0], x_range[1], y_range[0], y_range[1]))
plt.title(title)
ax.set_xlabel(r"$w_1$")
ax.set_ylabel(r"$w_2$")
plt.tight_layout()
return ax

sns.reset_orig()
_ = plot_curve(pathological_curve_loss, plot_3d=True)
plt.show()

In terms of optimization, you can image that $w_1$ and $w_2$ are weight parameters, and the curvature represents the loss surface over the space of $w_1$ and $w_2$. Note that in typical networks, we have many, many more parameters than two, and such curvatures can occur in multi-dimensional spaces as well.

Ideally, our optimization algorithm would find the center of the ravine and focuses on optimizing the parameters towards the direction of $w_2$. However, if we encounter a point along the ridges, the gradient is much greater in $w_1$ than $w_2$, and we might end up jumping from one side to the other. Due to the large gradients, we would have to reduce our learning rate slowing down learning significantly.
To test our algorithms, we can implement a simple function to train two parameters on such a surface:

```python
def train_curve(optimizer_func, curve_func=pathological_curve_loss, num_updates=100, init=[5,5]):
    """
    Inputs:
    optimizer_func - Constructor of the optimizer to use. Should only take a parameter list
    curve_func - Loss function (e.g. pathological curvature)
    num_updates - Number of updates/steps to take when optimizing
    init - Initial values of parameters. Must be a list/tuple with two elements representing w_1 and w_2
    Outputs:
    Numpy array of shape [num_updates, 3] with [t,:2] being the parameter values at step t, and [t,2] the loss at t.
    """
    weights = nn.Parameter(torch.FloatTensor(init), requires_grad=True)
    optim = optimizer_func([weights])
    list_points = []
    for _ in range(num_updates):
        loss = curve_func(weights[0], weights[1])
        list_points.append(torch.cat([weights.data.detach(), loss.unsqueeze(dim=0).detach()], dim=0).
        optim.zero_grad()
        loss.backward()
        optim.step()
    points = torch.stack(list_points, dim=0).numpy()
    return points
```

Next, let’s apply the different optimizers on our curvature. Note that we set a much higher learning rate for the optimization algorithms as you would in a standard neural network. This is because we only have 2 parameters instead of tens of thousands or even millions.

```python
SGD_points = train_curve(lambda params: SGD(params, lr=10))
SGDMom_points = train_curve(lambda params: SGDMomentum(params, lr=10, momentum=0.9))
Adam_points = train_curve(lambda params: Adam(params, lr=1))
```

To understand best how the different algorithms worked, we visualize the update step as a line plot through the loss surface. We will stick with a 2D representation for readability.

```python
all_points = np.concatenate([SGD_points, SGDMom_points, Adam_points], axis=0)
ax = plot_curve(pathological_curve_loss,
               x_range=(-np.abs(all_points[:,0]).max(), np.abs(all_points[:,0]).max()),
               y_range=(all_points[:,1].min(), all_points[:,1].max()),
               plot_3d=False)
ax.plot(SGD_points[:,0], SGD_points[:,1], color="red", marker="o", zorder=1, label="SGD")
ax.plot(SGDMom_points[:,0], SGDMom_points[:,1], color="blue", marker="o", zorder=2, label="SGDMom")
ax.plot(Adam_points[:,0], Adam_points[:,1], color="grey", marker="o", zorder=3, label="Adam")
plt.legend()
plt.show()
```
We can clearly see that SGD is not able to find the center of the optimization curve and has a problem converging due to the steep gradients in $w_1$. In contrast, Adam and SGD with momentum nicely converge as the changing direction of $w_1$ is canceling itself out. On such surfaces, it is crucial to use momentum.

**Steep optima**

A second type of challenging loss surfaces are steep optima. In those, we have a larger part of the surface having very small gradients while around the optimum, we have very large gradients. For instance, take the following loss surfaces:

```python
[32]: def bivar_gaussian(w1, w2, x_mean=0.0, y_mean=0.0, x_sig=1.0, y_sig=1.0):
    norm = 1 / (2 * np.pi * x_sig * y_sig)
    x_exp = (-1 * (w1 - x_mean)**2) / (2 * x_sig**2)
    y_exp = (-1 * (w2 - y_mean)**2) / (2 * y_sig**2)
    return norm * torch.exp(x_exp + y_exp)

def comb_func(w1, w2):
    z = -bivar_gaussian(w1, w2, x_mean=1.0, y_mean=-0.5, x_sig=0.2, y_sig=0.2)
    z -= bivar_gaussian(w1, w2, x_mean=-1.0, y_mean=0.5, x_sig=0.2, y_sig=0.2)
    z -= bivar_gaussian(w1, w2, x_mean=-0.5, y_mean=-0.8, x_sig=0.2, y_sig=0.2)
    return z

_ = plot_curve(comb_func, x_range=(-2,2), y_range=(-2,2), plot_3d=True, title="Steep optima")
```
Most of the loss surface has very little to no gradients. However, close to the optima, we have very steep gradients. To reach the minimum when starting in a region with lower gradients, we expect an adaptive learning rate to be crucial.

To verify this hypothesis, we can run our three optimizers on the surface:

```python
[33]: SGD_points = train_curve(lambda params: SGD(params, lr=.5), comb_func, init=[0,0])
SGDMom_points = train_curve(lambda params: SGDMomentum(params, lr=1, momentum=0.9), comb_func, init=[0,0])
Adam_points = train_curve(lambda params: Adam(params, lr=0.2), comb_func, init=[0,0])

all_points = np.concatenate([SGD_points, SGDMom_points, Adam_points], axis=0)
ax = plot_curve(comb_func, x_range=(-2, 2), y_range=(-2, 2), plot_3d=False, title="Steep optima")
ax.plot(SGD_points[:,0], SGD_points[:,1], color="red", marker="o", zorder=3, label="SGD", alpha=0.7)
ax.plot(SGDMom_points[:,0], SGDMom_points[:,1], color="blue", marker="o", zorder=2, label="SGDMom", alpha=0.7)
ax.plot(Adam_points[:,0], Adam_points[:,1], color="grey", marker="o", zorder=1, label="Adam", alpha=0.7)
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
plt.legend()
plt.show()
```
SGD first takes very small steps until it touches the border of the optimum. First reaching a point around \((-0.75, -0.5)\), the gradient direction has changed and pushes the parameters to \((0.8, 0.5)\) from which SGD cannot recover anymore (only with many, many steps). A similar problem has SGD with momentum, only that it continues the direction of the touch of the optimum. The gradients from this time step are so much larger than any other point that the momentum \(m_t\) is overpowered by it. Finally, Adam is able to converge in the optimum showing the importance of adaptive learning rates.

**What optimizer to take**

After seeing the results on optimization, what is our conclusion? Should we always use Adam and never look at SGD anymore? The short answer: no. There are many papers saying that in certain situations, SGD (with momentum) generalizes better where Adam often tends to overfit \([5,6]\). This is related to the idea of finding wider optima. For instance, see the illustration of different optima below (credit: Keskar et al., 2017):

The black line represents the training loss surface, while the dotted red line is the test loss. Finding sharp, narrow minima can be helpful for finding the minimal training loss. However, this doesn’t mean that it also minimizes the test loss as especially flat minima have shown to generalize better. You can imagine that the test dataset has a slightly shifted loss surface due to the different examples than in the training set. A small change can have a significant influence for sharp minima, while flat minima are generally more robust to this change.

In the next tutorial, we will see that some network types can still be better optimized with SGD and learning rate scheduling than Adam. Nevertheless, Adam is the most commonly used optimizer in Deep Learning as it usually performs better than other optimizers, especially for deep networks.
5.6.4 Conclusion

In this tutorial, we have looked at initialization and optimization techniques for neural networks. We have seen that a good initialization has to balance the preservation of the gradient variance as well as the activation variance. This can be achieved with the Xavier initialization for tanh-based networks, and the Kaiming initialization for ReLU-based networks. In optimization, concepts like momentum and adaptive learning rate can help with challenging loss surfaces but don’t guarantee an increase in performance for neural networks.

5.6.5 References


5.7 Tutorial 5: Inception, ResNet and DenseNet

Filled notebook:

Pre-trained models:

In this tutorial, we will implement and discuss variants of modern CNN architectures. There have been many different architectures been proposed over the past few years. Some of the most impactful ones, and still relevant today, are the following: GoogleNet/Inception architecture (winner of ILSVRC 2014), ResNet (winner of ILSVRC 2015), and DenseNet (best paper award CVPR 2017). All of them were state-of-the-art models when being proposed, and the core ideas of these networks are the foundations for most current state-of-the-art architectures. Thus, it is important to understand these architectures in detail and learn how to implement them.

Let’s start with importing our standard libraries here.

```python
# Standard libraries
import os
import numpy as np
import random
from PIL import Image
from types import SimpleNamespace

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

(continues on next page)
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

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

# Torchvision
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms

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

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

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

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

We also have pretrained models and Tensorboards (more on this later) for this tutorial, and download them below.

```
import urllib.request
from urllib.error import HTTPError

# Github URL where saved models are stored for this tutorial
# Files to download
˓
→ckpt",
        "tensorboards/GoogleNet/events.out.tfevents.googlenet",
        "tensorboards/ResNet/events.out.tfevents.resnet",
        "tensorboards/ResNetPreAct/events.out.tfevents.resnetpreact",
        "tensorboards/DenseNet/events.out.tfevents.densenet"]

# Create checkpoint path if it doesn’t exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)
```

(continues on next page)
# For each file, check whether it already exists. If not, try downloading it.

```python
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/",1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print("Downloading %s..." % file_url)
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive folder, or contact the author with the full output including the following error:

            ", e)
```

Throughout this tutorial, we will train and evaluate the models on the CIFAR10 dataset. This allows you to compare the results obtained here with the model you have implemented in the first assignment. As we have learned from the previous tutorial about initialization, it is important to have the data preprocessed with a zero mean. Therefore, as a first step, we will calculate the mean and standard deviation of the CIFAR dataset:

```python
[4]: train_dataset = CIFAR10(root=DATASET_PATH, train=True, download=True)
DATA_MEANS = (train_dataset.data / 255.0).mean(axis=(0,1,2))
DATA_STD = (train_dataset.data / 255.0).std(axis=(0,1,2))
print("Data mean", DATA_MEANS)
print("Data std", DATA_STD)
```

```
Files already downloaded and verified
Data mean [0.49139968 0.48215841 0.44653091]
Data std [0.24703223 0.24348513 0.26158784]
```

We will use this information to define a `transforms.Normalize` module which will normalize our data accordingly. Additionally, we will use data augmentation during training. This reduces the risk of overfitting and helps CNNs to generalize better. Specifically, we will apply two random augmentations.

First, we will flip each image horizontally by a chance of 50% (`transforms.RandomHorizontalFlip`). The object class usually does not change when flipping an image, and we don’t expect any image information to be dependent on the horizontal orientation. This would be however different if we would try to detect digits or letters in an image, as those have a certain orientation.

The second augmentation we use is called `transforms.RandomResizedCrop`. This transformation scales the image in a small range, while eventually changing the aspect ratio, and crops it afterward in the previous size. Therefore, the actual pixel values change while the content or overall semantics of the image stays the same.

We will randomly split the training dataset into a training and a validation set. The validation set will be used for determining early stopping. After finishing the training, we test the models on the CIFAR test set.

```python
[5]: test_transform = transforms.Compose([transforms.ToTensor(),
                                         transforms.Normalize(DATA_MEANS, DATA_STD)])

# For training, we add some augmentation. Networks are too powerful and would overfit.
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                      transforms.RandomResizedCrop((32,32),scale=(0.8,
                                      ratio=(0.9,1.1)),
                                      transforms.ToTensor(),
                                      transforms.Normalize(DATA_MEANS, DATA_STD)])
```

(continues on next page)
# Loading the training dataset. We need to split it into a training and validation part.

`train_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=train_transform, download=True)`

`val_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=test_transform, download=True)`

`set_seed(42)`

`train_set, _ = torch.utils.data.random_split(train_dataset, [45000, 5000])`

`set_seed(42)`

`_, val_set = torch.utils.data.random_split(val_dataset, [45000, 5000])`

# Loading the test set

`test_set = CIFAR10(root=DATASET_PATH, train=False, transform=test_transform, download=True)`

# We define a set of data loaders that we can use for various purposes later.

`train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True, pin_memory=True, num_workers=4)`

`val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False, drop_last=False, num_workers=4)`

`test_loader = data.DataLoader(test_set, batch_size=128, shuffle=False, drop_last=False, num_workers=4)`

To verify that our normalization works, we can print out the mean and standard deviation of the single batch. The mean should be close to 0 and the standard deviation close to 1 for each channel:

```python
imgs, _ = next(iter(train_loader))
print("Batch mean", imgs.mean(dim=[0,2,3]))
print("Batch std", imgs.std(dim=[0,2,3]))
```

```
Batch mean tensor([-0.0143, -0.0204, -0.0017])
Batch std tensor([0.9772, 0.9650, 0.9809])
```

Finally, let’s visualize a few images from the training set, and how they look like after random data augmentation:

```python
NUM_IMAGES = 4
images = [train_dataset[idx][0] for idx in range(NUM_IMAGES)]
orig_images = [Image.fromarray(train_dataset.data[idx]) for idx in range(NUM_IMAGES)]
orig_images = [test_transform(img) for img in orig_images]

img_grid = torchvision.utils.make_grid(torch.stack(images + orig_images, dim=0), nrow=4, normalize=True, pad_value=0.5)
img_grid = img_grid.permute(1, 2, 0)
plt.figure(figsize=(8,8))
plt.title("Augmentation examples on CIFAR10")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```
5.7.1 PyTorch Lightning

In this notebook and in many following ones, we will make use of the library PyTorch Lightning. PyTorch Lightning is a framework that simplifies your code needed to train, evaluate, and test a model in PyTorch. It also handles logging into TensorBoard, a visualization toolkit for ML experiments, and saving model checkpoints automatically with minimal code overhead from our side. This is extremely helpful for us as we want to focus on implementing different model architectures and spend little time on other code overhead. Note that at the time of writing/teaching, the framework has been released in version 1.0. Future versions might have a slightly changed interface and thus might not work perfectly with the code (we will try to keep it up-to-date as much as possible).

Now, we will take the first step in PyTorch Lightning, and continue to explore the framework in our other tutorials. First, we import the library:

![Augmentation examples on CIFAR10](image)

```
[8]: # PyTorch Lightning
    try:
        import pytorch_lightning as pl
    except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
        !pip install pytorch-lightning==1.0.3
        import pytorch_lightning as pl
```

PyTorch Lightning comes with a lot of useful functions, such as one for setting the seed:

```
[9]: # Setting the seed
    pl.seed_everything(42)
```

```
[9]: 42
```

Thus, in the future, we don’t have to define our own `set_seed` function anymore.

In PyTorch Lightning, we define `pl.LightningModule` (subtracting from `torch.nn.Module`) that organize our code into 5 main sections:
1. Initialization (**init**), where we create all necessary parameters/models

2. Optimizers (**configure_optimizers**) where we create the optimizers, learning rate scheduler, etc.

3. Training loop (**training_step**) where we only have to define the loss calculation for a single batch (the loop of optimizer.zero_grad(), loss.backward() and optimizer.step(), as well as any logging/saving operation, is done in the background)

4. Validation loop (**validation_step**) where similarly to the training, we only have to define what should happen per step

5. Test loop (**test_step**) which is the same as validation, only on a test set.

Therefore, we don’t abstract the PyTorch code, but rather organize it and define some default operations that are commonly used. If you need to change something else in your training/validation/test loop, there are many possible functions you can overwrite (see the docs for details).

Now we can look at an example of how a Lightning Module for training a CNN looks like:

```python
[10]: class CIFARTrainer(pl.LightningModule):
    def __init__(self, model_name, model_hparams, optimizer_name, optimizer_hparams):
        ""
        Inputs:
        model_name - Name of the model/CNN to run. Used for creating the model 
        (see function below)
        model_hparams - Hyperparameters for the model, as dictionary.
        optimizer_name - Name of the optimizer to use. Currently supported: Adam,
        optimizer_hparams - Hyperparameters for the optimizer, as dictionary.
        ""
        super().__init__()
        self.save_hyperparameters()
        self.model = create_model(model_name, model_hparams)
        self.loss_module = nn.CrossEntropyLoss()
        self.example_input_array = torch.zeros((1, 3, 32, 32), dtype=torch.float32)
        if self.hparams.optimizer_name == "Adam":
            optimizer = optim.AdamW(self.parameters(), **self.hparams.optimizer_hparams)
        elif self.hparams.optimizer_name == "SGD":
            optimizer = optim.SGD(self.parameters(), **self.hparams.optimizer_hparams)
        else:
            assert False, "Unknown optimizer: \"%s\" % self.hparams.optimizer_name
```

(continues on next page)
We will reduce the learning rate by 0.1 after 100 and 150 epochs.

```
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[100,150],
    gamma=0.1)
return [optimizer], [scheduler]
```

```
def training_step(self, batch, batch_idx):
    # "batch" is the output of the training data loader.
    imgs, labels = batch
    preds = self.model(imgs)
    loss = self.loss_module(preds, labels)
    acc = (preds.argmax(dim=-1) == labels).float().mean()

    self.log('train_acc', acc, on_step=False, on_epoch=True)  # Logs the accuracy per epoch to tensorboard (weighted average over batches)
    self.log('train_loss', loss)
    return loss  # Return tensor to call ".backward" on
```

```
def validation_step(self, batch, batch_idx):
    imgs, labels = batch
    preds = self.model(imgs).argmax(dim=-1)
    acc = (labels == preds).float().mean()
    self.log('val_acc', acc)  # By default logs it per epoch (weighted average over batches)
```

```
def test_step(self, batch, batch_idx):
    imgs, labels = batch
    preds = self.model(imgs).argmax(dim=-1)
    acc = (labels == preds).float().mean()
    self.log('test_acc', acc)  # By default logs it per epoch (weighted average over batches), and returns it afterwards
```

We see that the code is organized and clear, which helps if someone else tries to understand your code.

Another important part of PyTorch Lightning is the concept of callbacks. Callbacks are self-contained functions that contain the non-essential logic of your Lightning Module. They are usually called after finishing a training epoch, but can also influence other parts of your training loop. For instance, we will use the following two pre-defined callbacks: LearningRateMonitor and ModelCheckpoint. The learning rate monitor adds the current learning rate to our TensorBoard, which helps to verify that our learning rate scheduler works correctly. The model checkpoint callback allows you to customize the saving routine of your checkpoints. For instance, how many checkpoints to keep, when to save, which metric to look out for, etc. We import them below:

```
# Callbacks
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
```

To allow running multiple different models with the same Lightning module, we define a function below that maps a model name to the model class. At this stage, the dictionary `model_dict` is empty, but we will fill it throughout the notebook with our new models.

```
model_dict = {}
```

```
def create_model(model_name, model_hparams):
    if model_name in model_dict:
```
return model_dict[model_name](**model_hparams)
else:
    assert False, "Unknown model name \"%s\". Available models are: \"%s\" % (model_name, str(model_dict.keys()))

Similarly, to use the activation function as another hyperparameter in our model, we define a “name to function” dict below:

```
[13]: act_fn_by_name = {
    "tanh": nn.Tanh,
    "relu": nn.ReLU,
    "leakyrelu": nn.LeakyReLU,
    "gelu": nn.GELU
}
```

If we pass the classes or objects directly as an argument to the Lightning module, we couldn’t take advantage of PyTorch Lightning’s automatically hyperparameter saving and loading.

Besides the Lightning module, the second most important module in PyTorch Lightning is the Trainer. The trainer is responsible to execute the training steps defined in the Lightning module and completes the framework. Similar to the Lightning module, you can override any key part that you don’t want to be automated, but the default settings are often the best practice to do. For a full overview, see the documentation. The most important functions we use below are:

- **trainer.fit**: Takes as input a lightning module, a training dataset, and an (optional) validation dataset. This function trains the given module on the training dataset with occasional validation (default once per epoch, can be changed)

- **trainer.test**: Takes as input a model and a dataset on which we want to test. It returns the test metric on the dataset.

For training and testing, we don’t have to worry about things like setting the model to eval mode (`model.eval()`), as this is all done automatically. See below how we define a training function for our models:

```
[14]: def train_model(model_name, save_name=None, **kwargs):
    
    Inputs:
    model_name - Name of the model you want to run. Is used to look up the class in "model_dict"
    save_name (optional) - If specified, this name will be used for creating the checkpoint and logging directory.
    
    if save_name is None:
        save_name = model_name

    # Create a PyTorch Lightning trainer with the generation callback
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, save_name),
    # Where to save models
    checkpoint_callback=ModelCheckpoint(save_weights_only=True,
    #mode="max", monitor="val_acc"), # Save the best checkpoint based on the maximum val_acc recorded. Saves only weights and not optimizer
    gpus=1 if str(device)="cuda:0" else 0,
    # We run on a single GPU (if possible)
    max_epochs=180,
    # How many epochs to train for if no patience is set
    callbacks=[LearningRateMonitor("epoch")],
    # Log learning rate every epoch
```
Finally, we can focus on the Convolutional Neural Networks we want to implement today: GoogleNet, ResNet, and DenseNet.

### 5.7.2 Inception

The GoogleNet, proposed in 2014, won the ImageNet Challenge because of its usage of the Inception modules. In general, we will mainly focus on the concept of Inception in this tutorial instead of the specifics of the GoogleNet, as based on Inception, there have been many follow-up works (Inception-v2, Inception-v3, Inception-v4, Inception-ResNet,...). The follow-up works mainly focus on increasing efficiency and enabling very deep Inception networks. However, for a fundamental understanding, it is sufficient to look at the original Inception block.

An Inception block applies four convolution blocks separately on the same feature map: a 1x1, 3x3, and 5x5 convolution, and a max pool operation. This allows the network to look at the same data with different receptive fields. Of course, learning only 5x5 convolution would be theoretically more powerful. However, this is not only more computation and memory heavy but also tends to overfit much easier. The overall inception block looks like below (figure credit - Szegedy et al.):

The additional 1x1 convolutions before the 3x3 and 5x5 convolutions are used for dimensionality reduction. This is especially crucial as the feature maps of all branches are merged afterward, and we don’t want any explosion of feature size. As 5x5 convolutions are 25 times more expensive than 1x1 convolutions, we can save a lot of computation and parameters by reducing the dimensionality before the large convolutions.

We can now try to implement the Inception Block ourselves:

```python
[15]: class InceptionBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(InceptionBlock, self).__init__()

        # 1x1 convolution
        self.conv1x1 = nn.Conv2d(in_channels, out_channels // 4, kernel_size=1, stride=1, padding=0)

        # 3x3 convolution
        self.conv3x3 = nn.Conv2d(in_channels, out_channels // 4, kernel_size=3, stride=1, padding=1)

        # 5x5 convolution
        self.conv5x5 = nn.Conv2d(in_channels, out_channels // 4, kernel_size=5, stride=1, padding=2)

        # max pooling
        self.pool = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)

        # dimensionality reduction
        self.shortcut = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, padding=0)

    def forward(self, x):
        # 1x1 convolution branch
        x1x1 = self.conv1x1(x)

        # 3x3 convolution branch
        x3x3 = self.conv3x3(x)

        # 5x5 convolution branch
        x5x5 = self.conv5x5(x)

        # max pooling branch
        x_pool = self.pool(x)

        # dimensionality reduction
        x_shortcut = self.shortcut(x)

        return torch.cat([x1x1, x3x3, x5x5, x_pool, x_shortcut], 1)
```

(continues on next page)
```python
def __init__(self, c_in, c_red : dict, c_out : dict, act_fn):
    
    Inputs:
    c_in - Number of input feature maps from the previous layers
    c_red - Dictionary with keys "3x3" and "5x5" specifying the output of the
dimensionality reducing 1x1 convolutions
    c_out - Dictionary with keys "1x1", "3x3", "5x5", and "max"
    act_fn - Activation class constructor (e.g. nn.ReLU)
    
    super().__init__()
    # 1x1 convolution branch
    self.conv_1x1 = nn.Sequential(
        nn.Conv2d(c_in, c_out['1x1'], kernel_size=1),
        nn.BatchNorm2d(c_out['1x1']),
        act_fn()
    )
    # 3x3 convolution branch
    self.conv_3x3 = nn.Sequential(
        nn.Conv2d(c_in, c_red['3x3'], kernel_size=1),
        nn.BatchNorm2d(c_red['3x3']),
        act_fn(),
        nn.Conv2d(c_red['3x3'], c_out['3x3'], kernel_size=3, padding=1),
        nn.BatchNorm2d(c_out['3x3']),
        act_fn()
    )
    # 5x5 convolution branch
    self.conv_5x5 = nn.Sequential(
        nn.Conv2d(c_in, c_red['5x5'], kernel_size=1),
        nn.BatchNorm2d(c_red['5x5']),
        act_fn(),
        nn.Conv2d(c_red['5x5'], c_out['5x5'], kernel_size=5, padding=2),
        nn.BatchNorm2d(c_out['5x5']),
        act_fn()
    )
    # Max-pool branch
    self.max_pool = nn.Sequential(
        nn.MaxPool2d(kernel_size=3, padding=1, stride=1),
        nn.Conv2d(c_in, c_out['max'] , kernel_size=1),
        nn.BatchNorm2d(c_out['max']),
        act_fn()
    )

def forward(self, x):
    x_1x1 = self.conv_1x1(x)
    x_3x3 = self.conv_3x3(x)
    x_5x5 = self.conv_5x5(x)
    x_max = self.max_pool(x)
    x_out = torch.cat([x_1x1, x_3x3, x_5x5, x_max], dim=1)
    return x_out
```

The GoogleNet architecture consists of stacking multiple Inception blocks with occasional max pooling to reduce the height and width of the feature maps. The original GoogleNet was designed for image sizes of ImageNet (224x224
pixels) and had almost 7 million parameters. As we train on CIFAR10 with image sizes of 32x32, we don’t require such a heavy architecture, and instead, apply a reduced version. The number of channels for dimensionality reduction and output per filter (1x1, 3x3, 5x5, and max pooling) need to be manually specified and can be changed if interested. The general intuition is to have the most filters for the 3x3 convolutions, as they are powerful enough to take the context into account while requiring almost a third of the parameters of the 5x5 convolution.

```python
[16]: class GoogleNet(nn.Module):
    
    def __init__(self, num_classes=10, act_fn_name="relu", **kwargs):
        super().__init__()
        self.hparams = SimpleNamespace(num_classes=num_classes,
                                        act_fn_name=act_fn_name,
                                        act_fn=act_fn_by_name[act_fn_name])

        self._create_network()
        self._init_params()

    def _create_network(self):
        # A first convolution on the original image to scale up the channel size
        self.input_net = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            self.hparams.act_fn(),
        )

        # Stacking inception blocks
        self.inception_blocks = nn.Sequential(
            InceptionBlock(64, c_red={"3x3":32,"5x5":16}, c_out={"1x1":16,"3x3":32,
                                                                  "5x5":8,"max":8}, act_fn=self.hparams.act_fn),
            InceptionBlock(64, c_red={"3x3":32,"5x5":16}, c_out={"1x1":24,"3x3":48,
                                                                  "5x5":12,"max":12}, act_fn=self.hparams.act_fn),
            nn.MaxPool2d(3, stride=2, padding=1),  # 32x32 => 16x16
            InceptionBlock(96, c_red={"3x3":32,"5x5":16}, c_out={"1x1":24,"3x3":48,
                                                                  "5x5":12,"max":12}, act_fn=self.hparams.act_fn),
            InceptionBlock(96, c_red={"3x3":32,"5x5":16}, c_out={"1x1":16,"3x3":48,
                                                                  "5x5":16,"max":16}, act_fn=self.hparams.act_fn),
            InceptionBlock(96, c_red={"3x3":32,"5x5":16}, c_out={"1x1":16,"3x3":48,
                                                                  "5x5":16,"max":16}, act_fn=self.hparams.act_fn),
            InceptionBlock(96, c_red={"3x3":32,"5x5":16}, c_out={"1x1":32,"3x3":48,
                                                                  "5x5":24,"max":24}, act_fn=self.hparams.act_fn),
            nn.MaxPool2d(3, stride=2, padding=1),  # 16x16 => 8x8
            InceptionBlock(128, c_red={"3x3":48,"5x5":16}, c_out={"1x1":32,"3x3":64,
                                                                  "5x5":16,"max":16}, act_fn=self.hparams.act_fn),
            InceptionBlock(128, c_red={"3x3":48,"5x5":16}, c_out={"1x1":32,"3x3":64,
                                                                  "5x5":16,"max":16}, act_fn=self.hparams.act_fn),
        )

        # Mapping to classification output
        self.output_net = nn.Sequential(
            nn.AdaptiveAvgPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(128, self.hparams.num_classes)
        )

    def _init_params(self):
        # Based on our discussion in Tutorial 4, we should initialize the
        # convolutions according to the activation function
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
 nn.init.kaiming_normal_(m.weight, nonlinearity=self.hparams.act_fn_name)  

elif isinstance(m, nn.BatchNorm2d):  
    nn.init.constant_(m.weight, 1)  
    nn.init.constant_(m.bias, 0)  

def forward(self, x):
    x = self.input_net(x)
    x = self.inception_blocks(x)
    x = self.output_net(x)
    return x

Now, we can integrate our model to the model dictionary we defined above:

[17]: model_dict["GoogleNet"] = GoogleNet

The training of the model is handled by PyTorch Lightning, and we just have to define the command to start. Note that we train for almost 200 epochs, which takes about an hour on Lisa’s default GPUs (GTX1080Ti). We would recommend using the saved models and train your own model if you are interested.

[18]: googlenet_model, googlenet_results = train_model(model_name="GoogleNet",  
            model_hparams={"num_classes": 10,  
                           "act_fn_name": "relu"},  
            optimizer_name="Adam",  
            optimizer_hparams={"lr": 1e-3,  
                                "weight_decay": 1e-4})

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Found pretrained model at ../saved_models/tutorial5/GoogleNet.ckpt, loading...

We will compare the results later in the notebooks, but we can already print them here for a first glance:

[19]: print("GoogleNet Results", googlenet_results)

GoogleNet Results ('test': 0.8970000147819519, 'val': 0.9039999842643738)
Tensorboard log

A nice extra of PyTorch Lightning is the automatic logging into TensorBoard. To give you a better intuition of what TensorBoard can be used, we can look at the board that PyTorch Lightning has been generated when training the GoogleNet. TensorBoard provides an inline functionality for Jupyter notebooks, and we use it here:

```python
# Import tensorboard
from torch.utils.tensorboard import SummaryWriter
```

```python
%load_ext tensorboard
```

```bash
# Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH!
%tensorboard --logdir ../saved_models/tutorial5/tensorboards/GoogleNet/
```

TensorBoard is organized in multiple tabs. The main tab is the scalar tab where we can log the development of single numbers. For example, we have plotted the training loss, accuracy, learning rate, etc. If we look at the training or validation accuracy, we can really see the impact of using a learning rate scheduler. Reducing the learning rate gives our model a nice increase in training performance. Similarly, when looking at the training loss, we see a sudden decrease at this point. However, the high numbers on the training set compared to validation indicate that our model was overfitting which is inevitable for such large networks.

Another interesting tab in TensorBoard is the graph tab. It shows us the network architecture organized by building blocks from the input to the output. It basically shows the operations taken in the forward step of CIFARTrainer. Double-click on a module to open it. Feel free to explore the architecture from a different perspective. The graph visualization can often help you to validate that your model is actually doing what it is supposed to do, and you don’t miss any layers in the computation graph.
5.7.3 ResNet

The ResNet paper is one of the most cited AI papers, and has been the foundation for neural networks with more than 1,000 layers. Despite its simplicity, the idea of residual connections is highly effective as it supports stable gradient propagation through the network. Instead of modeling \( x_{l+1} = F(x_l) \), we model \( x_{l+1} = x_l + F(x_l) \) where \( F \) is a non-linear mapping (usually a sequence of NN modules like convolutions, activation functions, and normalizations). If we do backpropagation on such residual connections, we obtain:

\[
\frac{\partial x_{l+1}}{\partial x_l} = I + \frac{\partial F(x_l)}{\partial x_l}
\]

The bias towards the identity matrix guarantees a stable gradient propagation being less effected by \( F \) itself. There have been many variants of ResNet proposed, which mostly concern the function \( F \), or operations applied on the sum. In this tutorial, we look at two of them: the original ResNet block, and the Pre-Activation ResNet block. We visually compare the blocks below (figure credit - He et al.):

The original ResNet block applies a non-linear activation function, usually ReLU, after the skip connection. In contrast, the pre-activation ResNet block applies the non-linearity at the beginning of \( F \). Both have their advantages and disadvantages. For very deep network, however, the pre-activation ResNet has shown to perform better as the gradient flow is guaranteed to have the identity matrix as calculated above, and is not harmed by any non-linear activation applied to it. For comparison, in this notebook, we implement both ResNet types as shallow networks.

Let’s start with the original ResNet block. The visualization above already shows what layers are included in \( F \). One special case we have to handle is when we want to reduce the image dimensions in terms of width and height. The basic ResNet block requires \( F(x_l) \) to be of the same shape as \( x_l \). Thus, we need to change the dimensionality of \( x_l \) as well before adding to \( F(x_l) \). The original implementation used an identity mapping with stride 2 and padded additional feature dimensions with 0. However, the more common implementation is to use a 1x1 convolution with stride 2 as it allows us to change the feature dimensionality while being efficient in parameter and computation cost.

The code for the ResNet block is relatively simple, and shown below:

```python
[22]: class ResNetBlock(nn.Module):
    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        
        # Network representing F
        self.net = nn.Sequential(
            nn.Conv2d(c_in, c_out, kernel_size=3, padding=1, stride=1 if not subsample else 2, bias=False), # No bias needed as the Batch Norm handles it
            act_fn(),
            nn.Conv2d(c_out, c_out, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(c_out)
        )
```

(continues on next page)
# 1x1 convolution with stride 2 means we take the upper left value, and transform it to new output size

```python
self.downsample = nn.Conv2d(c_in, c_out, kernel_size=1, stride=2) if subsample else None
self.act_fn = act_fn()
```

```python
def forward(self, x):
    z = self.net(x)
    if self.downsample is not None:
        x = self.downsample(x)
    out = z + x
    out = self.act_fn(out)
    return out
```

The second block we implement is the pre-activation ResNet block. For this, we have to change the order of layer in `self.net`, and do not apply an activation function on the output. Additionally, the downsampling operation has to apply a non-linearity as well as the input, $x_i$, has not been processed by a non-linearity yet. Hence, the block looks as follows:

```python
class PreActResNetBlock(nn.Module):
    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        
        Inputs:
        c_in - Number of input features
        act_fn - Activation class constructor (e.g. nn.ReLU)
        subsample - If True, we want to apply a stride inside the block and reduce the output shape by 2 in height and width
        c_out - Number of output features. Note that this is only relevant if subsample is True, as otherwise, c_out = c_in

        super().__init__()

        if not subsample:
            c_out = c_in

        # Network representing F
        self.net = nn.Sequential(
            nn.BatchNorm2d(c_in),
            act_fn(),
            nn.Conv2d(c_in, c_out, kernel_size=3, padding=1, stride=1 if not subsample else 2, bias=False),
            nn.BatchNorm2d(c_out),
            act_fn(),
            nn.Conv2d(c_out, c_out, kernel_size=3, padding=1, bias=False)
        )

        # 1x1 convolution needs to apply non-linearity as well as not done on skip connection
        self.downsample = nn.Sequential(
            nn.BatchNorm2d(c_in),
            act_fn(),
            nn.Conv2d(c_in, c_out, kernel_size=1, stride=2, bias=False)
        ) if subsample else None
```

(continues on next page)
def forward(self, x):
    z = self.net(x)
    if self.downsample is not None:
        x = self.downsample(x)
    out = z + x
    return out

Similarly to the model selection, we define a dictionary to create a mapping from string to block class. We will use the string name as hyperparameter value in our model to choose between the ResNet blocks. Feel free to implement any other ResNet block type and add it here as well.

[24]:
```python
resnet_blocks_by_name = {
    "ResNetBlock": ResNetBlock,
    "PreActResNetBlock": PreActResNetBlock
}
```

The overall ResNet architecture consists of stacking multiple ResNet blocks, of which some are downsampling the input. When talking about ResNet blocks in the whole network, we usually group them by the same output shape. Hence, if we say the ResNet has [3, 3, 3] blocks, it means that we have 3 times a group of 3 ResNet blocks, where a subsampling is taking place in the fourth and seventh block. The ResNet with [3, 3, 3] blocks on CIFAR10 is visualized below.

The three groups operate on the resolutions $32 \times 32$, $16 \times 16$ and $8 \times 8$ respectively. The blocks in orange denote ResNet blocks with downsampling. The same notation is used by many other implementations such as in the torchvision library from PyTorch. Thus, our code looks as follows:

[25]:
```python
class ResNet(nn.Module):
    def __init__(self, num_classes=10, num_blocks=[3,3,3], c_hidden=[16,32,64], act_fn_name="relu", block_name="ResNetBlock", **kwargs):
        
        Inputs:
        num_classes - Number of classification outputs (10 for CIFAR10)
        num_blocks - List with the number of ResNet blocks to use. The first block of each group uses downsampling, except the first.
        c_hidden - List with the hidden dimensionalities in the different blocks. Usually multiplied by 2 the deeper we go.
        act_fn_name - Name of the activation function to use, looked up in "act_fn_by_name"
        block_name - Name of the ResNet block, looked up in "resnet_blocks_by_name"
        
        
    super().__init__()
    assert block_name in resnet_blocks_by_name
    self.hparams = SimpleNamespace(num_classes=num_classes,
                                 c_hidden=c_hidden,
                                 num_blocks=num_blocks,
                                 act_fn_name=act_fn_name,
                                 block_class=resnet_blocks_by_name[block_name])
                                 
    self._create_network()
    self._init_params()

    def _create_network(self):
```

(continues on next page)
c_hidden = self.hparams.c_hidden

# A first convolution on the original image to scale up the channel size
if self.hparams.block_class == PreActResNetBlock:
    # => Don't apply non-
    self.input_net = nn.Sequential(
        nn.Conv2d(3, c_hidden[0], kernel_size=3, padding=1, bias=False)
    )
else:
    self.input_net = nn.Sequential(
        nn.Conv2d(3, c_hidden[0], kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(c_hidden[0]),
        self.hparams.act_fn()
    )

# Creating the ResNet blocks
blocks = []
for block_idx, block_count in enumerate(self.hparams.num_blocks):
    for bc in range(block_count):
        subsample = (bc == 0 and block_idx > 0) # Subsample the first block
of each group, except the very first one.
        blocks.append(
            self.hparams.block_class(c_in=c_hidden[block_idx if not subsample]
            else (block_idx-1)),
            act_fn=self.hparams.act_fn,  
            subsample=subsample,
            c_out=c_hidden[block_idx])
    self.blocks = nn.Sequential(*blocks)

# Mapping to classification output
self.output_net = nn.Sequential(
    nn.AdaptiveAvgPool2d((1,1)),
    nn.Flatten(),
    nn.Linear(c_hidden[-1], self.hparams.num_classes)
)

def __init_params__(self):
    # Based on our discussion in Tutorial 4, we should initialize the
    # convolutions according to the activation function
    # Fan-out focuses on the gradient distribution, and is commonly used in
    # ResNets
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity=self.
            hparams.act_fn_name)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)

    return x
We also need to add the new ResNet class to our model dictionary:

```python
model_dict["ResNet"] = ResNet
```

Finally, we can train our ResNet models. One difference to the GoogleNet training is that we explicitly use SGD with Momentum as optimizer instead of Adam. Adam often leads to a slightly worse accuracy on plain, shallow ResNets. It is not 100% clear why Adam performs worse in this context, but one possible explanation is related to ResNet’s loss surface. ResNet has been shown to produce smoother loss surfaces than networks without skip connection (see Li et al., 2018 for details). A possible visualization of the loss surface with/out skip connections is below (figure credit - Li et al.):

The $x$ and $y$ axis shows a projection of the parameter space, and the $z$ axis shows the loss values achieved by different parameter values. On smooth surfaces like the one on the right, we might not require an adaptive learning rate as Adam provides. Instead, Adam can get stuck in local optima while SGD finds the wider minima that tend to generalize better. However, to answer this question in detail, we would need an extra tutorial because it is not easy to answer. For now, we conclude: for ResNet architectures, consider the optimizer to be an important hyperparameter, and try training with both Adam and SGD. Let’s train the model below with SGD:

```python
resnet_model, resnet_results = train_model(model_name="ResNet",
model_hparams={"num_classes": 10,
"c_hidden": [16,32,64],
"num_blocks": [3,3,3],
"act_fn_name": "relu"},
optimizer_name="SGD",
optimizer_hparams={"lr": 0.1,
"momentum": 0.9,
"weight_decay": 1e-4})
```

Let’s also train the pre-activation ResNet as comparison:

```python
resnetpreact_model, resnetpreact_results = train_model(model_name="ResNet",
model_hparams={"num_classes": 10,
"c_hidden": [16,32,64],
"num_blocks": [3,3,3],
"act_fn_name": "relu"},
optimizer_name="SGD",
optimizer_hparams={"lr": 0.1,
"momentum": 0.9,
"weight_decay": 1e-4})
```
Similarly to our GoogleNet model, we also have a TensorBoard log for the ResNet model. We can open it below.

```
%tensorboard --logdir ../saved_models/tutorial5/tensorboards/ResNet/
```
Feel free to explore the TensorBoard yourself, including the computation graph. In general, we can see that with SGD, the ResNet has a higher training loss than the GoogleNet in the first stage of the training. After reducing the learning rate however, the model achieves even higher validation accuracies. We compare the precise scores at the end of the notebook.

5.7.4 DenseNet

DenseNet is another architecture for enabling very deep neural networks and takes a slightly different perspective on residual connections. Instead of modeling the difference between layers, DenseNet considers residual connections as a possible way to reuse features across layers, removing any necessity to learn redundant feature maps. If we go deeper into the network, the model learns abstract features to recognize patterns. However, some complex patterns consist of a combination of abstract features (e.g. hand, face, etc.), and low-level features (e.g. edges, basic color, etc.). To find these low-level features in the deep layers, standard CNNs have to learn copy such feature maps, which wastes a lot of parameter complexity. DenseNet provides an efficient way of reusing features by having each convolution depends on all previous input features, but add only a small amount of filters to it. See the figure below for an illustration (figure credit - Hu et al.):
slightly different activations for the same features to different layers, depending on what is needed. Overall, we can implement it as follows:

```python
[30]: class DenseLayer(nn.Module):
    def __init__(self, c_in, bn_size, growth_rate, act_fn):
        """Inputs:
        c_in - Number of input channels
        bn_size - Bottleneck size (factor of growth rate) for the output of the 1x1 convolution. Typically between 2 and 4.
        growth_rate - Number of output channels of the 3x3 convolution
        act_fn - Activation class constructor (e.g. nn.ReLU)
        """
        super().__init__()
        self.net = nn.Sequential(
            nn.BatchNorm2d(c_in),
            act_fn(),
            nn.Conv2d(c_in, bn_size * growth_rate, kernel_size=1, bias=False),
            nn.BatchNorm2d(bn_size * growth_rate),
            act_fn(),
            nn.Conv2d(bn_size * growth_rate, growth_rate, kernel_size=3, padding=1, bias=False)
            )
    def forward(self, x):
        out = self.net(x)
        out = torch.cat([out, x], dim=1)
        return out
```

The module DenseBlock summarizes multiple dense layers applied in sequence. Each dense layer takes as input the original input concatenated with all previous layers’ feature maps:

```python
[31]: class DenseBlock(nn.Module):
    def __init__(self, c_in, num_layers, bn_size, growth_rate, act_fn):
        """Inputs:
        c_in - Number of input channels
        num_layers - Number of dense layers to apply in the block
        bn_size - Bottleneck size to use in the dense layers
        growth_rate - Growth rate to use in the dense layers
        act_fn - Activation function to use in the dense layers
        """
        super().__init__()
        layers = []
        for layer_idx in range(num_layers):
            layers.append(DenseLayer(c_in=c_in + layer_idx * growth_rate, # Input channels are original plus the feature maps from previous layers
                                bn_size=bn_size,
                                growth_rate=growth_rate,
                                act_fn=act_fn))
        self.block = nn.Sequential(*layers)
    def forward(self, x):
        out = self.block(x)
```

(continues on next page)
Finally, the TransitionLayer takes as input the final output of a dense block and reduces its channel dimensionality using a 1x1 convolution. To reduce the height and width dimension, we take a slightly different approach than in ResNet and apply an average pooling with kernel size 2 and stride 2. This is because we don’t have an additional connection to the output that would consider the full 2x2 patch instead of a single value. Besides, it is more parameter efficient than using a 3x3 convolution with stride 2. Thus, the layer is implemented as follows:

```python
[32]: class TransitionLayer(nn.Module):
    def __init__(self, c_in, c_out, act_fn):
        super().__init__()
        self.transition = nn.Sequential(
            nn.BatchNorm2d(c_in),
            act_fn(),
            nn.Conv2d(c_in, c_out, kernel_size=1, bias=False),
            nn.AvgPool2d(kernel_size=2, stride=2)  # Average the output for each 2x2 pixel group
        )
    def forward(self, x):
        return self.transition(x)
```

Now we can put everything together and create our DenseNet. To specify the number of layers, we use a similar notation as in ResNets and pass on a list of ints representing the number of layers per block. After each dense block except the last one, we apply a transition layer to reduce the dimensionality by 2.

```python
[33]: class DenseNet(nn.Module):
    def __init__(self, num_classes=10, num_layers=[6,6,6,6], bn_size=2, growth_rate=16, act_fn_name="relu", **kwargs):
        super().__init__()
        self.hparams = SimpleNamespace(num_classes=num_classes,
                                        num_layers=num_layers,
                                        bn_size=bn_size,
                                        growth_rate=growth_rate,
                                        act_fn_name=act_fn_name,
                                        act_fn=act_fn_by_name[act_fn_name])
        self._create_network()
        self._init_params()

    def _create_network(self):
        c_hidden = self.hparams.growth_rate * self.hparams.bn_size # The start number of hidden channels

        self.input_net = nn.Sequential(
            nn.Conv2d(3, c_hidden, kernel_size=3, padding=1)  # No batch norm or activation function as done inside the Dense layers
        )

        blocks = []
        for block_idx, num_layers in enumerate(self.hparams.num_layers):
            blocks.append(DenseBlock(c_in=c_hidden,
```

(continues on next page)
Let’s also add the DenseNet to our model dictionary:

```python
model_dict["DenseNet"] = DenseNet
```

Lastly, we train our network. In contrast to ResNet, DenseNet does not show any issues with Adam, and hence we train it with this optimizer. The other hyperparameters are chosen to result in a network with a similar parameter size as the ResNet and GoogleNet. Commonly, when designing very deep networks, DenseNet is more parameter efficient than ResNet while achieving a similar or even better performance.

```python
densenet_model, densenet_results = train_model(model_name="DenseNet",
(continues on next page)
model_hparams={"num_classes": 10,
    "num_layers": [6, 6, 6, 6],
    "bn_size": 2,
    "growth_rate": 16,
    "act_fn_name": "relu"},
optimizer_name="Adam",
optimizer_hparams={"lr": 1e-3,
    "weight_decay": 1e-
˓→4})

GPU available: True, used: True
I1109 15:33:10.534986 139927971620672 distributed.py:49] GPU available: True, used:
˓→True
TPU available: False, using: 0 TPU cores
I1109 15:33:10.536422 139927971620672 distributed.py:49] TPU available: False, using:
˓→0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
I1109 15:33:10.538130 139927971620672 accelerator_connector.py:385] LOCAL_RANK: 0 -
˓→CUDA_VISIBLE_DEVICES: [0]

Found pretrained model at ../saved_models/tutorial5/DenseNet.ckpt, loading...

HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',
    layout=Layout(flex='2'), max=...)

HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing',
    layout=Layout(flex='2'), max=...)

**Tensorboard log**

Finally, we also have another TensorBoard for the DenseNet training. We take a look at it below:

```
[36]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH! Feel free
˓→to change "ResNet" to "ResNetPreAct"
%tensorboard --logdir ../saved_models/tutorial5/tensorboards/DenseNet/
```
The overall course of the validation accuracy and training loss resemble the training of GoogleNet, which is also related to training the network with Adam. Feel free to explore the training metrics yourself.

### 5.7.5 Conclusion and Comparison

After discussing each model separately, and training all of them, we can finally compare them. First, let’s organize the results of all models in a table:

```python
import tabulate
from IPython.display import display, HTML
all_models = [
    ("GoogleNet", googlenet_results, googlenet_model),
    ("ResNet", resnet_results, resnet_model),
    ("ResNetPreAct", resnetpreact_results, resnetpreact_model),
    ("DenseNet", densenet_results, densenet_model)
]
table = [ [model_name,
            "%.2f%%" % (100.0*model_results["val"]),
            "%.2f%%" % (100.0*model_results["test"])
        ]
```
First of all, we see that all models are performing reasonably well. Simple models as you have implemented them in the practical achieve considerably lower performance, which is beside the lower number of parameters also attributed to the architecture design choice. GoogleNet is the model to obtain the lowest performance on the validation and test set, although it is very close to DenseNet. A proper hyperparameter search over all the channel sizes in GoogleNet would likely improve the accuracy of the model to a similar level, but this is also expensive given a large number of hyperparameters. ResNet outperforms both DenseNet and GoogleNet by more than 1% on the validation set, while there is a minor difference between both versions, original and pre-activation. We can conclude that for shallow networks, the place of the activation function does not seem to be crucial, although papers have reported the contrary for very deep networks (e.g. He et al.).

In general, we can conclude that ResNet is a simple, but powerful architecture. If we would apply the models on more complex tasks with larger images and more layers inside the networks, we would likely see a bigger gap between GoogleNet and skip-connection architectures like ResNet and DenseNet. A comparison with deeper models on CIFAR10 can be for example found here. Interestingly, DenseNet outperforms the original ResNet on their setup but comes closely behind the Pre-Activation ResNet. The best model, a Dual Path Network (Chen et. al), is actually a combination of ResNet and DenseNet showing that both offer different advantages.

**Which model should I choose for my task?**

We have reviewed four different models. So, which one should we choose if have given a new task? Usually, starting with a ResNet is a good idea given the superior performance of the CIFAR dataset and its simple implementation. Besides, for the parameter number we have chosen here, ResNet is the fastest as DenseNet and GoogleNet have many more layers that are applied in sequence in our primitive implementation. However, if you have a really difficult task, such as semantic segmentation on HD images, more complex variants of ResNet and DenseNet are recommended.

### 5.8 Tutorial 6: Transformers and Multi-Head Attention

**Filled notebook:**

**Pre-trained models:**

In this tutorial, we will discuss one of the most impactful architectures of the last 2 years: the Transformer model. Since the paper *Attention Is All You Need* by Vaswani et al. had been published in 2017, the Transformer architecture has continued to beat benchmarks in many domains, most importantly in Natural Language Processing. Transformers with an incredible amount of parameters can generate long, convincing essays, and opened up new application fields of AI. As the hype of the Transformer architecture seems not to come to an end in the next years, it is important to understand how it works, and have implemented it yourself, which we will do in this notebook.

Despite the huge success of Transformers in NLP, we will *not* include the NLP domain in our notebook here. Why? Firstly, the Master AI at UvA offers many great NLP courses that will take a closer look at the application of the Transformer architecture in NLP (NLP2, Advanced Topics in Computational Semantics). Secondly, assignment 2 takes already a closer look at language generation on character level, on which you could easily apply our transformer architecture. Finally, and most importantly, there is so much more to the Transformer architecture. NLP is the domain the Transformer architecture has been originally proposed for and had the greatest impact on, but it also accelerated
research in other domains, recently even Computer Vision. Thus, we focus here on what makes the Transformer and self-attention so powerful in general.

Below, we import our standard libraries. As in Tutorial 5, we will use PyTorch Lightning as an additional framework. If you are not familiar with PyTorch Lightning, please make sure to have read Tutorial 5 carefully.

```python
## Standard libraries
import os
import numpy as np
import random
import math
import json
from functools import partial

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

## tqdm for loading bars
from tqdm.notebook import tqdm

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

## Torchvision
import torchvision
from torchvision.datasets import CIFAR100
from torchvision import transforms

## PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install pytorch-lightning==1.0.3
import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial6"

# Setting the seed
pl.seed_everything(42)
```
(continues on next page)
# Ensure that all operations are deterministic on GPU (if used) for reproducibility

```python
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", device)

Device: cuda:0

Two pre-trained models are downloaded below. Make sure to have adjusted your CHECKPOINT_PATH before running this code if not already done.

```python
import urllib.request
from urllib.error import HTTPError

# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial6/"

# Files to download
pretrained_files = ["ReverseTask.ckpt", "SetAnomalyTask.ckpt"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/",1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print("Downloading %s..." % file_url)
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive folder, or contact the author with the full output including the following error:

```

5.8.1 The Transformer architecture

In the first part of this notebook, we will implement the Transformer architecture by hand. As the architecture is so popular, there already exists a Pytorch module `nn.Transformer` (documentation) and a tutorial on how to use it for next token prediction. However, we will implement it here ourselves, to get through to the smallest details.

There are of course many more tutorials out there about attention and Transformers. Below, we list a few that are worth exploring if you are interested in the topic and might want yet another perspective on the topic after this one:

- **Transformer: A Novel Neural Network Architecture for Language Understanding** (Jakob Uszkoreit, 2017) - The original Google blog post about the Transformer paper, focusing on the application in machine translation.
- **The Illustrated Transformer** (Jay Alammar, 2018) - A very popular and great blog post intuitively explaining the Transformer architecture with many nice visualizations. The focus is on NLP.
- **Attention? Attention!** (Lilian Weng, 2018) - A nice blog post summarizing attention mechanisms in many domains including vision.
- **Illustrated: Self-Attention** (Raimi Karim, 2019) - A nice visualization of the steps of self-attention. Recommended going through if the explanation below is too abstract for you.
The Transformer family (Lilian Weng, 2020) - A very detailed blog post reviewing more variants of Transformers besides the original one.

What is Attention?

The attention mechanism describes a recent new group of layers in neural networks that has attracted a lot of interest in the past few years, especially in sequence tasks. There are a lot of different possible definitions of “attention” in the literature, but the one we will use here is the following: the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements’ keys. So what does this exactly mean? The goal is to take an average over the features of multiple elements. However, instead of weighting each element equally, we want to weight them depending on their actual values. In other words, we want to dynamically decide on which inputs we want to “attend” more than others. In particular, an attention mechanism has usually four parts we need to specify:

- **Query**: The query is a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.

- **Keys**: For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is “offering”, or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.

- **Values**: For each input element, we also have a value vector. This feature vector is the one we want to average over.

- **Score function**: To rate which elements we want to pay attention to, we need to specify a score function $f_{\text{attn}}$. The score function takes the query and a key as input, and output the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product, or a small MLP.

The weights of the average are calculated by a softmax over all score function outputs. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query. If we try to describe it with pseudo-math, we can write:

$$a_i = \frac{\exp(f_{\text{attn}}(\text{key}_i, \text{query}))}{\sum_j \exp(f_{\text{attn}}(\text{key}_j, \text{query}))}, \quad \text{out} = \sum_i a_i \cdot \text{value}_i$$

Visually, we can show the attention over a sequence of words as follows:

For every word, we have one key and one value vector. The query is compared to all keys with a score function (in this case the dot product) to determine the weights. The softmax is not visualized for simplicity. Finally, the value vectors of all words are averaged using the attention weights.

Most attention mechanisms differ in terms of what queries they use, how the key and value vectors are defined, and what score function is used. The attention applied inside the Transformer architecture is called **self-attention**. In self-attention, each sequence element provides a key, value, and query. For each element, we perform an attention layer where based on its query, we check the similarity of the all sequence elements’ keys, and returned a different, averaged value vector for each element. We will now go into a bit more detail by first looking at the specific implementation of the attention mechanism which is in the Transformer case the scaled dot product attention.
Scaled Dot Product Attention

The core concept behind self-attention is the scaled dot product attention. Our goal is to have an attention mechanism with which any element in a sequence can attend to any other while still being efficient to compute. The dot product attention takes as input a set of queries $Q \in \mathbb{R}^{T \times d_k}$, keys $K \in \mathbb{R}^{T \times d_k}$ and values $V \in \mathbb{R}^{T \times d_v}$ where $T$ is the sequence length, and $d_k$ and $d_v$ are the hidden dimensionality for queries/keys and values respectively. For simplicity, we neglect the batch dimension for now. The attention value from element $i$ to $j$ is based on its similarity of the query $Q_i$ and key $K_j$, using the dot product as the similarity metric. In math, we calculate the dot product attention as follows:

$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$

The matrix multiplication $QK^T$ performs the dot product for every possible pair of queries and keys, resulting in a matrix of the shape $T \times T$. Each row represents the attention logits for a specific element $i$ to all other elements in the sequence. On these, we apply a softmax and multiply with the value vector to obtain a weighted mean (the weights being determined by the attention). Another perspective on this attention mechanism offers the computation graph which is visualized below (figure credit - Vaswani et al., 2017).

One aspect we haven’t discussed yet is the scaling factor of $1/\sqrt{d_k}$. This scaling factor is crucial to maintain an appropriate variance of attention values after initialization. Remember that we initialize our layers with the intention of having equal variance throughout the model, and hence, $Q$ and $K$ might also have a variance close to 1. However, performing a dot product over two vectors with a variance $\sigma$ results in a scalar having $d_k$-times higher variance:

$$q_i \sim \mathcal{N}(0, \sigma), k_i \sim \mathcal{N}(0, \sigma) \rightarrow \text{Var} \left( \sum_{i=1}^{d_k} q_i \cdot k_i \right) = \sigma \cdot d_k$$

If we do not scale down the variance back to $\sigma$, the softmax over the logits will already saturate to 1 for one random element and 0 for all others. The gradients through the softmax will be close to zero so that we can’t learn the parameters appropriately.

The block Mask (opt.) in the diagram above represents the optional masking of specific entries in the attention matrix. This is for instance used if we stack multiple sequences with different lengths into a batch. To still benefit from parallelization in PyTorch, we pad the sentences to the same length and mask out the padding tokens during the calculation of the attention values. This is usually done by setting the respective attention logits to a very low value.

After we have discussed the details of the scaled dot product attention block, we can write a function below which computes the output features given the triple of queries, keys, and values:

```python
[3]:
def scaled_dot_product(q, k, v, mask=None):
    d_k = q.size()[-1]
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits = attn_logits / math.sqrt(d_k)
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -9e15)
    attention = F.softmax(attn_logits, dim=-1)
    values = torch.matmul(attention, v)
    return values, attention
```

Note that our code above supports any additional dimensionality in front of the sequence length so that we can also use it for batches. However, for a better understanding, let’s generate a few random queries, keys, and value vectors, and calculate the attention outputs:

```python
[4]:
seq_len, d_k = 3, 2
pl.seed_everything(42)
```
q = torch.randn(seq_len, d_k)
k = torch.randn(seq_len, d_k)
v = torch.randn(seq_len, d_k)
values, attention = scaled_dot_product(q, k, v)
print("Q

\[
\begin{bmatrix}
0.3367, & 0.1288, \\
0.2345, & 0.2303, \\
-1.1229, & -0.1863
\end{bmatrix}
\]
\)
print("K

\[
\begin{bmatrix}
2.2082, & -0.6380, \\
0.4617, & 0.2674, \\
0.5349, & 0.8094
\end{bmatrix}
\]
\)
print("V

\[
\begin{bmatrix}
1.1103, & -1.6898, \\
-0.9890, & 0.9580, \\
1.3221, & 0.8172
\end{bmatrix}
\]
\)
print("Values

\[
\begin{bmatrix}
0.5698, & -0.1520, \\
0.5379, & -0.0265, \\
0.2246, & 0.5556
\end{bmatrix}
\]
\)
print("Attention

\[
\begin{bmatrix}
0.4028, & 0.2886, & 0.3086, \\
0.3538, & 0.3069, & 0.3393, \\
0.1303, & 0.4630, & 0.4071
\end{bmatrix}
\]
\)

Before continuing, make sure you can follow the calculation of the specific values here, and also check it by hand. It is important to fully understand how the scaled dot product attention is calculated.

**Multi-Head Attention**

The scaled dot product attention allows a network to attend over a sequence. However, often there are multiple different aspects a sequence element wants to attend to, and a single weighted average is not a good option for it. This is why we extend the attention mechanisms to multiple heads, i.e. multiple different query-key-value triplets on the same features. Specifically, given a query, key, and value matrix, we transform those into $h$ sub-queries, sub-keys, and sub-values, which we pass through the scaled dot product attention independently. Afterward, we concatenate the heads and combine them with a final weight matrix. Mathematically, we can express this operation as:

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

where $\text{head}_i = \text{Attention}(QW^Q_i, KW^K_i, VW^V_i)$

We refer to this as Multi-Head Attention layer with the learnable parameters $W^Q_{1...h} \in \mathbb{R}^{D \times d_k}$, $W^K_{1...h} \in \mathbb{R}^{D \times d_k}$, $W^V_{1...h} \in \mathbb{R}^{D \times d_v}$, and $W^O \in \mathbb{R}^{h \cdot d_k \times d_{out}}$ ($D$ being the input dimensionality). Expressed in a computational graph, we can visualize it as below (figure credit - Vaswani et al., 2017).

How are we applying a Multi-Head Attention layer in a neural network, where we don’t have an arbitrary query, key, and value vector as input? Looking at the computation graph above, a simple but effective implementation is to set the current feature map in a NN, $X \in \mathbb{R}^{B \times T \times d_{model}}$, as $Q$, $K$ and $V$ ($B$ being the batch size, $T$ the sequence length, $d_{model}$ the hidden dimensionality of $X$). The consecutive weight matrices $W^Q$, $W^K$, and $W^V$ can transform $X$ to the
corresponding feature vectors that represent the queries, keys, and values of the input. Using this approach, we can implement the Multi-Head Attention module below.

```python
[5]: class MultiheadAttention(nn.Module):
    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()
        assert embed_dim % num_heads == 0, "Embedding dimension must be 0 modulo number of heads."
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads
        # Stack all weight matrices 1...h together for efficiency
        # Note that in many implementations you see "bias=False" which is optional
        self.qkv_proj = nn.Linear(input_dim, 3*embed_dim)
        self.o_proj = nn.Linear(embed_dim, embed_dim)
        self._reset_parameters()

    def _reset_parameters(self):
        # Original Transformer initialization, see PyTorch documentation
        nn.init.xavier_uniform_(self.qkv_proj.weight)
        self.qkv_proj.bias.data.fill_(0)
        nn.init.xavier_uniform_(self.o_proj.weight)
        self.o_proj.bias.data.fill_(0)

    def forward(self, x, mask=None, return_attention=False):
        batch_size, seq_length, embed_dim = x.size()
        qkv = self.qkv_proj(x)
        # Separate Q, K, V from linear output
        qkv = qkv.reshape(batch_size, seq_length, self.num_heads, 3*self.head_dim)
        qkv = qkv.permute(0, 2, 1, 3)  # [Batch, Head, SeqLen, Dims]
        q, k, v = qkv.chunk(3, dim=-1)
        # Determine value outputs
        values, attention = scaled_dot_product(q, k, v, mask=mask)
        values = values.reshape(batch_size, seq_length, embed_dim)
        o = self.o_proj(values)

        if return_attention:
            return o, attention
        else:
            return o
```

One crucial characteristic of the multi-head attention is that it is permutation-equivariant with respect to its inputs. This means that if we switch two input elements in the sequence, e.g. $X_1 \leftrightarrow X_2$ (neglecting the batch dimension for now), the output is exactly the same besides the elements 1 and 2 switched. Hence, the multi-head attention is actually looking at the input not as a sequence, but as a set of elements. This property makes the multi-head attention block and the Transformer architecture so powerful and widely applicable! But what if the order of the input is actually important for solving the task, like language modeling? The answer is to encode the position in the input features, which we will take a closer look at later (topic *Positional encodings* below).
Before moving on to creating the Transformer architecture, we can compare the self-attention operation with our other common layer competitors for sequence data: convolutions and recurrent neural networks. Below you can find a table by Vaswani et al. (2017) on the complexity per layer, the number of sequential operations, and maximum path length. The complexity is measured by the upper bound of the number of operations to perform, while the maximum path length represents the maximum number of steps a forward or backward signal has to traverse to reach any other position. The lower this length, the better gradient signals can backpropagate for long-range dependencies. Let’s take a look at the table below:

\( n \) is the sequence length, \( d \) is the representation dimension and \( k \) is the kernel size of convolutions. In contrast to recurrent networks, the self-attention layer can parallelize all its operations making it much faster to execute for smaller sequence lengths. However, when the sequence length exceeds the hidden dimensionality, self-attention becomes more expensive than RNNs. One way of reducing the computational cost for long sequences is by restricting the self-attention to a neighborhood of inputs to attend over, denoted by \( r \). Nevertheless, there has been recently a lot of work on more efficient Transformer architectures that still allow long dependencies, of which you can find an overview in the paper by Tay et al. (2020) if interested.

Transformer Encoder

Next, we will look at how to apply the multi-head attention blog inside the Transformer architecture. Originally, the Transformer model was designed for machine translation. Hence, it got an encoder-decoder structure where the encoder takes as input the sentence in the original language and generates an attention-based representation. On the other hand, the decoder attends over the encoded information and generates the translated sentence in an autoregressive manner, as in a standard RNN. While this structure is extremely useful for Sequence-to-Sequence tasks with the necessity of autoregressive decoding, we will focus here on the encoder part. Many advances in NLP have been made using pure encoder-based Transformer models (if interested, models include the BERT-family, the Vision Transformer, and more), and in our tutorial, we will also mainly focus on the encoder part. If you have understood the encoder architecture, the decoder is a very small step to implement as well. The full Transformer architecture looks as follows (figure credit - Vaswani et al., 2017):

The encoder consists of \( N \) identical blocks that are applied in sequence. Taking as input \( x \), it is first passed through a Multi-Head Attention block as we have implemented above. The output is added to the original input using a residual connection, and we apply a consecutive Layer Normalization on the sum. Overall, it calculates LayerNorm\((x + \text{Multihead}(x, x, x))\) (\( x \) being \( Q, K \) and \( V \) input to the attention layer). The residual connection is crucial in the Transformer architecture for two reasons:

1. Similar to ResNets, Transformers are designed to be very deep. Some models contain more than 24 blocks in the encoder. Hence, the residual connections are crucial for enabling a smooth gradient flow through the model.
2. Without the residual connection, the information about the original sequence is lost. Remember that the Multi-Head Attention layer ignores the position of elements in a sequence, and can only learn it based on the input features. Removing the residual connections would mean that this information is lost after the first attention layer (after initialization), and with a randomly initialized query and key vector, the output vectors for position \( i \) has no relation to its original input. All outputs of the attention are likely to represent similar/same information, and there is no chance for the model to distinguish which information came from which input element. An alternative option to residual connection would be to fix at least one head to focus on its original input, but this is very inefficient and does not have the benefit of the improved gradient flow.

The Layer Normalization also plays an important role in the Transformer architecture as it enables faster training and provides small regularization. Additionally, it ensures that the features are in a similar magnitude among the elements in the sequence. We are not using Batch Normalization because it depends on the batch size which is often small with Transformers (they require a lot of GPU memory), and BatchNorm has shown to perform particularly bad in language as the features of words tend to have a much higher variance (there are many, very rare words which need to be considered for a good distribution estimate).
Additionally to the Multi-Head Attention, a small fully connected feed-forward network is added to the model, which is applied to each position separately and identically. Specifically, the model uses a Linear → ReLU → Linear MLP. The full transformation including the residual connection can be expressed as:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

$$x = \text{LayerNorm}(x + \text{FFN}(x))$$

This MLP adds extra complexity to the model and allows transformations on each sequence element separately. You can imagine as this allows the model to “post-process” the new information added by the previous Multi-Head Attention, and prepare it for the next attention block. Usually, the inner dimensionality of the MLP is 2-8 × larger than $d_{\text{model}}$, i.e. the dimensionality of the original input $x$. The general advantage of a wider layer instead of a narrow, multi-layer MLP is the faster, parallelizable execution.

Finally, after looking at all parts of the encoder architecture, we can start implementing it below. We first start by implementing a single encoder block. Additionally to the layers described above, we will add dropout layers in the MLP and on the output of the MLP and Multi-Head Attention for regularization.

```python
[6]: class EncoderBlock(nn.Module):

def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
    # Inputs:
    # input_dim - Dimensionality of the input
    # num_heads - Number of heads to use in the attention block
    # dim_feedforward - Dimensionality of the hidden layer in the MLP
    # dropout - Dropout probability to use in the dropout layers
    super().__init__()

    # Attention layer
    self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)

    # Two-layer MLP
    self.linear_net = nn.Sequential(
        nn.Linear(input_dim, dim_feedforward),
        nn.Dropout(dropout),
        nn.ReLU(inplace=True),
        nn.Linear(dim_feedforward, input_dim)
    )

    # Layers to apply in between the main layers
    self.norm1 = nn.LayerNorm(input_dim)
    self.norm2 = nn.LayerNorm(input_dim)
    self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Attention part
        attn_out = self.self_attn(x, mask=mask)
        x = x + self.dropout(attn_out)
        x = self.norm1(x)

        # MLP part
        linear_out = self.linear_net(x)
        x = x + self.dropout(linear_out)
        x = self.norm2(x)

        return x
```

5.8. Tutorial 6: Transformers and Multi-Head Attention
Based on this block, we can implement a module for the full Transformer encoder. Additionally to a forward function that iterates through the sequence of encoder blocks, we also provide a function called `get_attention_maps`. The idea of this function is to return the attention probabilities for all Multi-Head Attention blocks in the encoder. This helps us in understanding, and in a sense, explaining the model. However, the attention probabilities should be interpreted with a grain of salt as it does not necessarily reflect the true interpretation of the model (there is a series of papers about this, including Attention is not Explanation and Attention is not not Explanation).

```python
[7]:
class TransformerEncoder(nn.Module):
    def __init__(self, num_layers, **block_args):
        super().__init__()
        self.layers = nn.ModuleList([EncoderBlock(**block_args) for _ in range(num_layers)])

    def forward(self, x, mask=None):
        for l in self.layers:
            x = l(x, mask=mask)
        return x

    def get_attention_maps(self, x, mask=None):
        attention_maps = []
        for l in self.layers:
            _, attn_map = l.self_attn(x, mask=mask, return_attention=True)
            attention_maps.append(attn_map)
            x = l(x)
        return attention_maps
```

### Positional encoding

We have discussed before that the Multi-Head Attention block is permutation-equivariant, and cannot distinguish whether an input comes before another one in the sequence or not. In tasks like language understanding, however, the position is important for interpreting the input words. The position information can therefore be added via the input features. We could learn a embedding for every possible position, but this would not generalize to a dynamical input sequence length. Hence, the better option is to use feature patterns that the network can identify from the features and potentially generalize to larger sequences. The specific pattern chosen by Vaswani et al. are sine and cosine functions of different frequencies, as follows:

\[
PE_{(\text{pos}, i)} = \begin{cases} 
\sin \left( \frac{\text{pos}}{10000^{i/\text{model}}} \right) & \text{if } i \mod 2 = 0 \\
\cos \left( \frac{\text{pos}}{10000^{(1-i)/\text{model}}} \right) & \text{otherwise}
\end{cases}
\]

`PE_{(\text{pos}, i)}` represents the position encoding at position `pos` in the sequence, and hidden dimensionality `i`. These values, concatenated for all hidden dimensions, are added to the original input features (in the Transformer visualization above, see “Positional encoding”), and constitute the position information. We distinguish between even (`i \mod 2 = 0`) and uneven (`i \mod 2 = 1`) hidden dimensionalities where we apply a sine/cosine respectively. The intuition behind this encoding is that you can represent `PE_{(\text{pos}+k, :)}` as a linear function of `PE_{(\text{pos}, :)}`, which might allow the model to easily attend to relative positions. The wavelengths in different dimensions range from $2\pi$ to $10000 \cdot 2\pi$.

The positional encoding is implemented below. The code is taken from the PyTorch tutorial about Transformers on NLP and adjusted for our purposes.

```python
[8]:
class PositionalEncoding(nn.Module):
    (continues on next page)
```
def __init__(self, d_model, max_len=5000):
    
    Inputs
    d_model - Hidden dimensionality of the input.
    max_len - Maximum length of a sequence to expect.
    
    super().__init__()

    # Create matrix of [SeqLen, HiddenDim] representing the positional encoding
    for max_len inputs
    pe = torch.zeros(max_len, d_model)
    position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    pe = pe.unsqueeze(0)

    self.register_buffer('pe', pe, persistent=False)

def forward(self, x):
    x = x + self.pe[:, :x.size(1)]
    return x

To understand the positional encoding, we can visualize it below. We will generate an image of the positional encoding over hidden dimensionality and position in a sequence. Each pixel, therefore, represents the change of the input feature we perform to encode the specific position. Let’s do it below.

[9]:
encod_block = PositionalEncoding(d_model=48, max_len=96)
pe = encod_block.pe.squeeze().T.cpu().numpy()

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,3))
pos = ax.imshow(pe, cmap="RdGy", extent=(1,pe.shape[1]+1,pe.shape[0]+1,1))
fig.colorbar(pos, ax=ax)
ax.set_xlabel("Position in sequence")
ax.set_ylabel("Hidden dimension")
ax.set_title("Positional encoding over hidden dimensions")
ax.set_xticks([1]+[i*10 for i in range(1,1+pe.shape[1]//10)])
ax.set_yticks([1]+[i*10 for i in range(1,1+pe.shape[0]//10)])
plt.show()
You can clearly see the sine and cosine waves with different wavelengths that encode the position in the hidden dimensions. Specifically, we can look at the sine/cosine wave for each hidden dimension separately, to get a better intuition of the pattern. Below we visualize the positional encoding for the hidden dimensions 1, 2, 3 and 4.

```python
[10]: sns.set_theme()
    fig, ax = plt.subplots(2, 2, figsize=(12,4))
    ax = [a for a_list in ax for a in a_list]
    for i in range(len(ax)):
        ax[i].plot(np.arange(1,17), pe[i,:16], color='C%i' % i, marker="o", markersize=6,
                   markeredgecolor="black")
        ax[i].set_title("Encoding in hidden dimension %i" % (i+1))
        ax[i].set_xlabel("Position in sequence", fontsize=10)
        ax[i].set_ylabel("Positional encoding", fontsize=10)
        ax[i].set_xticks(np.arange(1,17))
        ax[i].tick_params(axis='both', which='major', labelsize=10)
        ax[i].tick_params(axis='both', which='minor', labelsize=8)
        ax[i].set_ylim(-1.2, 1.2)
    fig.subplots_adjust(hspace=0.8)
    sns.reset_orig()
    plt.show()
```
As we can see, the patterns between the hidden dimension 1 and 2 only differ in the starting angle. The wavelength is $2\pi$, hence the repetition after position 6. The hidden dimensions 2 and 3 have about twice the wavelength.

**Learning rate warm-up**

One commonly used technique for training a Transformer is learning rate warm-up. This means that we gradually increase the learning rate from 0 on to our originally specified learning rate in the first few iterations. Thus, we slowly start learning instead of taking very large steps from the beginning. In fact, training a deep Transformer without learning rate warm-up can make the model diverge and achieve a much worse performance on training and testing. Take for instance the following plot by Liu et al. (2019) comparing Adam-vanilla (i.e. Adam without warm-up) vs Adam with a warm-up:

Clearly, the warm-up is a crucial hyperparameter in the Transformer architecture. Why is it so important? There are currently two common explanations. Firstly, Adam uses the bias correction factors which however can lead to a higher variance in the adaptive learning rate during the first iterations. Improved optimizers like RAdam have been shown to overcome this issue, not requiring warm-up for training Transformers. Secondly, the iteratively applied Layer Normalization across layers can lead to very high gradients during the first iterations, which can be solved by using Pre-Layer Normalization (similar to Pre-Activation ResNet), or replacing Layer Normalization by other techniques (Adaptive Normalization, Power Normalization).

Nevertheless, many applications and papers still use the original Transformer architecture with Adam, because warm-up is a simple, yet effective way of solving the gradient problem in the first iterations. There are many different schedulers we could use. For instance, the original Transformer paper used an exponential decay scheduler with a warm-up. However, the currently most popular scheduler is the cosine warm-up scheduler, which combines warm-up with a cosine-shaped learning rate decay. We can implement it below, and visualize the learning rate factor over epochs.

```python
class CosineWarmupScheduler(optim.lr_scheduler._LRScheduler):
    def __init__(self, optimizer, warmup, max_iters):
        self.warmup = warmup
        self.max_num_iters = max_iters
        super().__init__(optimizer)

    def get_lr(self):
        lr_factor = self.get_lr_factor(epoch=self.last_epoch)
```

(continues on next page)
```
return [base_lr * lr_factor for base_lr in self.base_lrs]

def get_lr_factor(self, epoch):
    lr_factor = 0.5 * (1 + np.cos(np.pi * epoch / self.max_num_iters))
    if epoch <= self.warmup:
        lr_factor *= epoch * 1.0 / self.warmup
    return lr_factor
```

[12]: # Needed for initializing the lr scheduler
    p = nn.Parameter(torch.empty(4,4))
    optimizer = optim.Adam([p], lr=1e-3)
    lr_scheduler = CosineWarmupScheduler(optimizer=optimizer, warmup=100, max_iters=2000)

    # Plotting
    epochs = list(range(2000))
    sns.set()
    plt.figure(figsize=(8,3))
    plt.plot(epochs, [lr_scheduler.get_lr_factor(e) for e in epochs])
    plt.ylabel("Learning rate factor")
    plt.xlabel("Iterations (in batches)")
    plt.title("Cosine Warm-up Learning Rate Scheduler")
    plt.show()
    sns.reset_orig()

In the first 100 iterations, we increase the learning rate factor from 0 to 1, whereas for all later iterations, we decay it using the cosine wave. Pre-implementations of this scheduler can be found in the popular NLP Transformer library huggingface.
Finally, we can embed the Transformer architecture into a PyTorch lightning module. From Tutorial 5, you know that PyTorch Lightning simplifies our training and test code, as well as structures the code nicely in separate functions. We will implement a template for a classifier based on the Transformer encoder. Thereby, we have a prediction output per sequence element. If we would need a classifier over the whole sequence, the common approach is to add an additional [CLS] token to the sequence, representing the classifier token. However, here we focus on tasks where we have an output per element.

Additionally to the Transformer architecture, we add a small input network (maps input dimensions to model dimensions), the positional encoding, and an output network (transforms output encodings to predictions). We also add the learning rate scheduler, which takes a step each iteration instead of once per epoch. This is needed for the warmup and the smooth cosine decay. The training, validation, and test step is left empty for now and will be filled for our task-specific models.

```
[13]: class TransformerPredictor(pl.LightningModule):

def __init__(self, input_dim, model_dim, num_classes, num_heads, num_layers, lr, 
warmup, max_iters, dropout=0.0, input_dropout=0.0):

    Inputs:
    input_dim - Hidden dimensionality of the input
    model_dim - Hidden dimensionality to use inside the Transformer
    num_classes - Number of classes to predict per sequence element
    num_heads - Number of heads to use in the Multi-Head Attention blocks
    num_layers - Number of encoder blocks to use.
    lr - Learning rate in the optimizer
    warmup - Number of warmup steps. Usually between 50 and 500
    max_iters - Number of maximum iterations the model is trained for. This
is needed for the CosineWarmup scheduler
    dropout - Dropout to apply inside the model
    input_dropout - Dropout to apply on the input features

    super().__init__()
    self.save_hyperparameters()
    self._create_model()

def _create_model(self):
    # Input_dim -> Model dim
    self.input_net = nn.Sequential(
        nn.Dropout(self.hparams.input_dropout),
        nn.Linear(self.hparams.input_dim, self.hparams.model_dim)
    )
    # Positional encoding for sequences
    self.positional_encoding = PositionalEncoding(d_model=self.hparams.model_dim)
    # Transformer
    self.transformer = TransformerEncoder(num_layers=self.hparams.num_layers, 
        input_dim=self.hparams.model_dim, 
        dim_feedforward=2*self.hparams.model_dim,
        num_heads=self.hparams.num_heads,
        dropout=self.hparams.dropout)
    # Output classifier per sequence element
    self.output_net = nn.Sequential(
        nn.Linear(self.hparams.model_dim, self.hparams.model_dim),
    )
    (continues on next page)
nn.LayerNorm(self.hparams.model_dim),
n.ReLU(inplace=True),
n.Dropout(self.hparams.dropout),
n.Linear(self.hparams.model_dim, self.hparams.num_classes)
)

    def forward(self, x, mask=None, add_positional_encoding=True):
        '''
        Inputs:
        x - Input features of shape [Batch, SeqLen, input_dim]
        mask - Mask to apply on the attention outputs (optional)
        add_positional_encoding - If True, we add the positional encoding to the
        input.
        '''
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        x = self.transformer(x, mask=mask)
        x = self.output_net(x)
        return x

    @torch.no_grad()
    def get_attention_maps(self, x, mask=None, add_positional_encoding=True):
        '''
        Function for extracting the attention matrices of the whole Transformer for a
        single batch.
        Input arguments same as the forward pass.
        '''
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        attention_maps = self.transformer.get_attention_maps(x, mask=mask)
        return attention_maps

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=self.hparams.lr)
        # We don't return the lr scheduler because we need to apply it per iteration, not per epoch
        self.lr_scheduler = CosineWarmupScheduler(optimizer,
                                                   warmup=self.hparams.warmup,
                                                   max_iters=self.hparams.max_iters)
        return optimizer

    def optimizer_step(self, *args, **kwargs):
        super().optimizer_step(*args, **kwargs)
        self.lr_scheduler.step() # Step per iteration

    def training_step(self, batch, batch_idx):
        raise NotimplementedError
def validation_step(self, batch, batch_idx):
    raise NotImplementedError

def test_step(self, batch, batch_idx):
    raise NotImplementedError

5.8.2 Experiments

After having finished the implementation of the Transformer architecture, we can start experimenting and apply it to various tasks. In this notebook, we will focus on two tasks: parallel Sequence-to-Sequence, and set anomaly detection. The two tasks focus on different properties of the Transformer architecture, and we go through them below.

Sequence to Sequence

A Sequence-to-Sequence task represents a task where the input and the output is a sequence, not necessarily of the same length. Popular tasks in this domain include machine translation and summarization. For this, we usually have a Transformer encoder for interpreting the input sequence, and a decoder for generating the output in an autoregressive manner. Here, however, we will go back to a much simpler example task and use only the encoder. Given a sequence of \( N \) numbers between 0 and \( M \), the task is to reverse the input sequence. In Numpy notation, if our input is \( x \), the output should be \( x[::-1] \). Although this task sounds very simple, RNNs can have issues with such because the task requires long-term dependencies. Transformers are built to support such, and hence, we expect it to perform very well.

First, let’s create a dataset class below.

```
class ReverseDataset(data.Dataset):
    def __init__(self, num_categories, seq_len, size):
        super().__init__()
        self.num_categories = num_categories
        self.seq_len = seq_len
        self.size = size
        self.data = torch.randint(self.num_categories, size=(self.size, self.seq_len))

    def __len__(self):
        return self.size

    def __getitem__(self, idx):
        inp_data = self.data[idx]
        labels = torch.flip(inp_data, dims=(0,))
        return inp_data, labels
```

We create an arbitrary number of random sequences of numbers between 0 and \( num\_categories-1 \). The label is simply the tensor flipped over the sequence dimension. We can create the corresponding data loaders below.

```
dataset = partial(ReverseDataset, 10, 16)
train_loader = data.DataLoader(dataset(50000), batch_size=128, shuffle=True, drop_last=True, pin_memory=True)
```

(continues on next page)
Let's look at an arbitrary sample of the dataset:

```python
[16]: inp_data, labels = train_loader.dataset[0]
    print("Input data:", inp_data)
    print("Labels: ", labels)
Input data: tensor([9, 6, 2, 0, 6, 2, 7, 9, 7, 3, 3, 4, 3, 7, 0, 9])
Labels:    tensor([9, 0, 7, 3, 4, 3, 3, 7, 9, 7, 2, 6, 0, 2, 6, 9])
```

During training, we pass the input sequence through the Transformer encoder and predict the output for each input token. We use the standard Cross-Entropy loss to perform this. Every number is represented as a one-hot vector. Remember that representing the categories as single scalars decreases the expressiveness of the model extremely as 0 and 1 are not closer related than 0 and 9 in our example. An alternative to a one-hot vector is using a learned embedding vector as it is provided by the PyTorch module `nn.Embedding`. However, using a one-hot vector with an additional linear layer as in our case has the same effect as an embedding layer (`self.input_net` maps one-hot vector to a dense vector, where each row of the weight matrix represents the embedding for a specific category).

To implement the training dynamic, we create a new class inheriting from `TransformerPredictor` and overwriting the training, validation and test step functions.

```python
[17]: class ReversePredictor(TransformerPredictor):
    def __calculate_loss(self, batch, mode="train"):
        # Fetch data and transform categories to one-hot vectors
        inp_data, labels = batch
        inp_data = F.one_hot(inp_data, num_classes=self.hparams.num_classes).float()

        # Perform prediction and calculate loss and accuracy
        preds = self.forward(inp_data, add_positional_encoding=True)
        loss = F.cross_entropy(preds.view(-1,preds.size(-1)), labels.view(-1))
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        # Logging
        self.log("%s_loss" % mode, loss)
        self.log("%s_acc" % mode, acc)
        return loss, acc

    def training_step(self, batch, batch_idx):
        loss, _ = self.__calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        _ = self.__calculate_loss(batch, mode="val")

    def test_step(self, batch, batch_idx):
        _ = self.__calculate_loss(batch, mode="test")
```

Finally, we can create a training function similar to the one we have seen in Tutorial 5 for PyTorch Lightning. We create a `pl.Trainer` object, running for $N$ epochs, logging in TensorBoard, and saving our best model based on the validation. Afterward, we test our models on the test set. An additional parameter we pass to the trainer here is `gradient_clip_val`. This clips the norm of the gradients for all parameters before taking an optimizer step and prevents the model from diverging if we obtain very high gradients at, for instance, sharp loss surfaces (see many good blog posts on gradient clipping, like DeepAI glossary). For Transformers, gradient clipping can help to further stabilize the training during the first few iterations, and also afterward. In plain PyTorch, you can apply gradient clipping via...
torch.nn.utils.clip_grad_norm_(...) (see documentation). The clip value is usually between 0.5 and 10, depending on how harsh you want to clip large gradients. After having explained this, let’s implement the training function:

```python
[18]: def train_reverse(**kwargs):
    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "ReverseTask")
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                         checkpoint_callback=ModelCheckpoint(save_weights_only=True, 
                         mode="max", monitor="val_acc"),
                         gpus=1 if str(device).startswith("cuda") else 0,
                         max_epochs=10,
                         gradient_clip_val=5,
                         progress_bar_refresh_rate=1)
    trainer.logger._default_hp_metric = None  # Optional logging argument that we don’t need
    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "ReverseTask.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = ReversePredictor.load_from_checkpoint(pretrained_filename)
    else:
        model = ReversePredictor(max_iters=trainer.max_epochs*len(train_loader),
                        **kwargs)
    trainer.fit(model, train_loader, val_loader)
    # Test best model on validation and test set
    val_result = trainer.test(model, test_dataloaders=val_loader, verbose=False)
    test_result = trainer.test(model, test_dataloaders=test_loader, verbose=False)
    result = {"test_acc": test_result[0]["test_acc"], "val_acc": val_result[0]["test_acc"]}
    return model, result
```

Finally, we can train the model. In this setup, we will use a single encoder block and a single head in the Multi-Head Attention. This is chosen because of the simplicity of the task, and in this case, the attention can actually be interpreted as an “explanation” of the predictions (compared to the other papers above dealing with deep Transformers).

```python
[19]: reverse_model, reverse_result = train_reverse(input_dim=train_loader.dataset.num_categories,
                                            model_dim=32,
                                            num_heads=1,
                                            num_classes=train_loader.dataset.num_categories,
                                            num_layers=1,
                                            dropout=0.0,
                                            lr=5e-4,
                                            warmup=50)
```

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]  

Found pretrained model, loading...
/home/phillip/anaconda3/envs/nlp1/lib/python3.7/site-packages/pytorch_lightning/utilities/distributed.py:45: UserWarning: The dataloader, test dataloader 0, does not have many workers which may be a bottleneck. Consider increasing the value of the `num_workers` argument (try 16 which is the number of cpus on this machine) in the `DataLoader` init to improve performance.
The warning of PyTorch Lightning regarding the number of workers can be ignored for now. As the data set is so simple and the `__getitem__` function finishes a negligible time, we don’t need subprocesses to provide us the data (in fact, more workers can slow down the training as we have communication overhead among processes/threads). First, let’s print the results:

```python
print("Val accuracy: \$4.2f\%" % (100.0 * reverse_result["val_acc"]))
print("Test accuracy: \$4.2f\%" % (100.0 * reverse_result["test_acc"]))

Val accuracy: 100.00%
Test accuracy: 100.00%
```

As we would have expected, the Transformer can correctly solve the task. However, how does the attention in the Multi-Head Attention block looks like for an arbitrary input? Let’s try to visualize it below.

```python
data_input, labels = next(iter(val_loader))
inp_data = F.one_hot(data_input, num_classes=reverse_model.hparams.num_classes).float()
inp_data = inp_data.to(device)
attention_maps = reverse_model.get_attention_maps(inp_data)
```

The object `attention_maps` is a list of length $N$ where $N$ is the number of layers. Each element is a tensor of shape `[Batch, Heads, SeqLen, SeqLen]`, which we can verify below.

```python
attention_maps[0].shape
```

```python
torch.Size([128, 1, 16, 16])
```

Next, we will write a plotting function that takes as input the sequences, attention maps, and an index indicating for which batch element we want to visualize the attention map. We will create a plot where over rows, we have different layers, while over columns, we show the different heads. Remember that the softmax has been applied for each row separately.

```python
def plot_attention_maps(input_data, attn_maps, idx=0):
    if input_data is not None:
        input_data = input_data[idx].detach().cpu().numpy()
    else:
        input_data = np.arange(attn_maps[0][idx].shape[-1])
    attn_maps = [m[idx].detach().cpu().numpy() for m in attn_maps]

    num_heads = attn_maps[0].shape[0]
    num_layers = len(attn_maps)
    seq_len = input_data.shape[0]
    fig_size = 4 if num_heads == 1 else 3
    fig, ax = plt.subplots(num_layers, num_heads, figsize=(num_heads*fig_size, num_layers*fig_size))
    if num_layers == 1:
        pass
    else:
        pass
```
ax = [ax]
if num_heads == 1:
    ax = [[a] for a in ax]
for row in range(num_layers):
    for column in range(num_heads):
        ax[row][column].imshow(attn_maps[row][column], origin='lower', vmin=0)
        ax[row][column].set_xticks(list(range(seq_len)))
        ax[row][column].set_xticklabels(input_data.tolist())
        ax[row][column].set_yticks(list(range(seq_len)))
        ax[row][column].set_yticklabels(input_data.tolist())
        ax[row][column].set_title("Layer %i, Head %i" % (row+1, column+1))
fig.subplots_adjust(hspace=0.5)
plt.show()

Finally, we can plot the attention map of our trained Transformer on the reverse task:

```
[24]: plot_attention_maps(data_input, attention_maps, idx=0)
```

![Layer 1, Head 1](image)

The model has learned to attend to the token that is on the flipped index of itself. Hence, it actually does what we intended it to do. We see that it however also pays some attention to values close to the flipped index. This is because the model doesn’t need the perfect, hard attention to solve this problem, but is fine with this approximate, noisy attention map. The close-by indices are caused by the similarity of the positional encoding, which we also intended with the positional encoding.
Set Anomaly Detection

Besides sequences, sets are another data structure that is relevant for many applications. In contrast to sequences, elements are unordered in a set. RNNs can only be applied on sets by assuming an order in the data, which however biases the model towards a non-existing order in the data. Vinyals et al. (2015) and other papers have shown that the assumed order can have a significant impact on the model’s performance, and hence, we should try to not use RNNs on sets. Ideally, our model should be permutation-equivariant/invariant such that the output is the same no matter how we sort the elements in a set.

Transformers offer the perfect architecture for this as the Multi-Head Attention is permutation-equivariant, and thus, outputs the same values no matter in what order we enter the inputs (inputs and outputs are permuted equally). The task we are looking at for sets is Set Anomaly Detection which means that we try to find the element(s) in a set that does not fit the others. In the research community, the common application of anomaly detection is performed on a set of images, where $N - 1$ images belong to the same category/have the same high-level features while one belongs to another category. Note that category does not necessarily have to relate to a class in a standard classification problem, but could be the combination of multiple features. For instance, on a face dataset, this could be people with glasses, male, beard, etc. An example of distinguishing different animals can be seen below. The first four images show foxes, while the last represents a different animal. We want to recognize that the last image shows a different animal, but it is not relevant which class of animal it is.

In this tutorial, we will use the CIFAR100 dataset. CIFAR100 has 600 images for 100 classes each with a resolution of 32x32, similar to CIFAR10. The larger amount of classes requires the model to attend to specific features in the images instead of coarse features as in CIFAR10, therefore making the task harder. We will show the model a set of 9 images of one class, and 1 image from another class. The task is to find the image that is from a different class than the other images. Using the raw images directly as input to the Transformer is not a good idea, because it is not translation invariant as a CNN, and would need to learn to detect image features from high-dimensional input first of all. Instead, we will use a pre-trained ResNet34 model from the torchvision package to obtain high-level, low-dimensional features of the images. The ResNet model has been pre-trained on the ImageNet dataset which contains 1 million images of 1k classes and varying resolutions. However, during training and testing, the images are usually scaled to a resolution of 224x224, and hence we rescale our CIFAR images to this resolution as well. Below, we will load the dataset, and prepare the data for being processed by the ResNet model.

```python
# ImageNet statistics
DATA_MEANS = np.array([0.485, 0.456, 0.406])
DATA_STD = np.array([0.229, 0.224, 0.225])
# As torch tensors for later preprocessing
TORCH_DATA_MEANS = torch.from_numpy(DATA_MEANS).view(1,3,1,1)
TORCH_DATA_STD = torch.from_numpy(DATA_STD).view(1,3,1,1)
# Resize to 224x224, and normalize to ImageNet statistic
transform = transforms.Compose([transforms.Resize((224,224)),
                                transforms.ToTensor(),
                                transforms.Normalize(DATA_MEANS, DATA_STD)])
# Loading the training dataset.
train_set = CIFAR100(root=DATASET_PATH, train=True, transform=transform,
                     download=True)
```
Next, we want to run the pre-trained ResNet model on the images, and extract the features before the classification layer. These are the most high-level features, and should sufficiently describe the images. CIFAR100 has some similarity to ImageNet, and thus we are not retraining the ResNet model in any form. However, if you would want to get the best performance and have a very large dataset, it would be better to add the ResNet to the computation graph during training and finetune its parameters as well. As we don’t have a large enough dataset and want to train our model efficiently, we will extract the features beforehand. Let’s load and prepare the model below.

```python
import os
os.environ["TORCH_HOME"] = CHECKPOINT_PATH
pretrained_model = torchvision.models.resnet34(pretrained=True)
# Remove classification layer
# In some models, it is called "fc", others have "classifier"
# Setting both to an empty sequential represents an identity map of the final features.
pretrained_model.fc = nn.Sequential()
pretrained_model.classifier = nn.Sequential()
# To GPU
pretrained_model = pretrained_model.to(device)
# Only eval, no gradient required
pretrained_model.eval()
for p in pretrained_model.parameters():
    p.requires_grad = False
```

We will now write an extraction function for the features below. This cell requires access to a GPU, as the model is rather deep and the images relatively large. The GPUs on GoogleColab are sufficient, but running this cell can take 2-3 minutes. Once it is run, the features are exported on disk so they don’t have to be recalculated every time you run the notebook. However, this requires >150MB free disk space. So it is recommended to run this only on a local computer if you have enough free disk and a GPU (GoogleColab is fine for this). If you do not have a GPU, you can download the features from the GoogleDrive folder.

```python
@torch.no_grad()
def extract_features(dataset, save_file):
    if not os.path.isfile(save_file):
        data_loader = data.DataLoader(dataset, batch_size=128, shuffle=False, drop_last=False, num_workers=4)
        extracted_features = []
        for imgs, _ in tqdm(data_loader):
            imgs = imgs.to(device)
            feats = pretrained_model(imgs)
            extracted_features.append(feats)
        extracted_features = torch.cat(extracted_features, dim=0)
        extracted_features = extracted_features.detach().cpu()
        torch.save(extracted_features, save_file)
    else:
        extracted_features = torch.load(save_file)
    return extracted_features
```
Let’s verify the feature shapes below. The training should have 50k elements, and the test 10k images. The feature dimension is 512 for the ResNet34. If you experiment with other models, you likely see a different feature dimension.

```
[28]: print("Train:", train_set_feats.shape)
print("Test: ", test_feats.shape)
Train: torch.Size([50000, 512])
Test: torch.Size([10000, 512])
```

As usual, we want to create a validation set to detect when we should stop training. In this case, we will split the training set into 90% training, 10% validation. However, the difficulty is here that we need to ensure that the validation set has the same number of images for all 100 labels. Otherwise, we have a class imbalance which is not good for creating the image sets. Hence, we take 10% of the images for each class, and move them into the validation set. The code below does exactly this.

```
## Split train into train+val
# Get labels from train set
labels = train_set.targets
# Get indices of images per class
labels = torch.LongTensor(labels)
num_labels = labels.max()+1
sorted_indices = torch.argsort(labels).reshape(num_labels, -1)  # [classes, num_imgs→per class]

# Determine number of validation images per class
num_val_exmps = sorted_indices.shape[1] // 10

# Get image indices for validation and training
val_indices = sorted_indices[:,:num_val_exmps].reshape(-1)
train_indices = sorted_indices[:,num_val_exmps:].reshape(-1)

# Group corresponding image features and labels
train_feats, train_labels = train_set_feats[train_indices], labels[train_indices]
val_feats, val_labels = train_set_feats[val_indices], labels[val_indices]
```

Now we can prepare a dataset class for the set anomaly task. We define an epoch to be the sequence in which each image has been exactly once as an “anomaly”. Hence, the length of the dataset is the number of images in it. For the training set, each time we access an item with `__getitem__`, we sample a random, different class than the image at the corresponding index `idx` has. In a second step, we sample \( N - 1 \) images of this sampled class. The set of 10 images is finally returned. The randomness in the `__getitem__` allows us to see a slightly different set during each iteration. However, we can’t use the same strategy for the test set as we want the test dataset to be the same every time we iterate over it. Hence, we sample the sets in the `__init__` method, and return those in `__getitem__`. The code below implements exactly this dynamic.

```
[30]: class SetAnomalyDataset(data.Dataset):
    def __init__(self, img_feats, labels, set_size=10, train=True):
        """
        Inputs:
        ```
img_feats - Tensor of shape [num_imgs, img_dim]. Represents the high-
level features.
labels - Tensor of shape [num_imgs], containing the class labels for the
images
set_size - Number of elements in a set. N-1 are sampled from one class,
and one from another one.
train - If True, a new set will be sampled every time __getitem__ is
called.

```python
super().__init__()
self.img_feats = img_feats
self.labels = labels
self.set_size = set_size-1  # The set size is here the size of correct images
self.train = train

# Tensors with indices of the images per class
self.num_labels = labels.max()+1
self.img_idx_by_label = torch.argsort(self.labels).reshape(self.num_labels, -1)

if not train:
    self.test_sets = self._create_test_sets()

def _create_test_sets(self):
    # Pre-generates the sets for each image for the test set
test_sets = []
    num_imgs = self.img_feats.shape[0]
    np.random.seed(42)
test_sets = [self.sample_img_set(self.labels[idx]) for idx in range(num_imgs)]
test_sets = torch.stack(test_sets, dim=0)
return test_sets

def sample_img_set(self, anomaly_label):
    # Samples a new set of images, given the label of the anomaly.
The sampled images come from a different class than anomaly_label
    # Sample class from 0,...,num_classes-1 while skipping anomaly_label as class
    set_label = np.random.randint(self.num_labels-1)
    if set_label >= anomaly_label:
        set_label += 1
    # Sample images from the class determined above
    img_indices = np.random.choice(self.img_idx_by_label.shape[1], size=self.set_size, replace=False)
    img_indices = self.img_idx_by_label[set_label, img_indices]
return img_indices

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

def __getitem__(self, idx):
    anomaly = self.img_feats[idx]
if self.train:  # If train => sample
    img_indices = self.sample_img_set(self.labels[idx])
else:  # If test => use pre-generated ones
    img_indices = self.test_sets[idx]

# Concatenate images. The anomaly is always the last image for simplicity
img_set = torch.cat([self.img_feats[img_indices], anomaly[None]], dim=0)
indices = torch.cat([img_indices, torch.LongTensor([idx])], dim=0)
label = img_set.shape[0]-1

    # We return the indices of the images for visualization purpose. "Label" is → the index of the anomaly
    return img_set, indices, label

Next, we can setup our datasets and data loaders below. Here, we will use a set size of 10, i.e. 9 images from one category + 1 anomaly. Feel free to change it if you want to experiment with the sizes.

[31]:
SET_SIZE = 10
test_labels = torch.LongTensor(test_set.targets)

train_anom_dataset = SetAnomalyDataset(train_feats, train_labels, set_size=SET_SIZE,
   train=True)
val_anom_dataset = SetAnomalyDataset(val_feats, val_labels, set_size=SET_SIZE,
   train=False)

train_anom_loader = data.DataLoader(train_anom_dataset, batch_size=64, shuffle=True,
   drop_last=True, num_workers=4, pin_memory=True)

val_anom_loader = data.DataLoader(val_anom_dataset, batch_size=64, shuffle=False,
   drop_last=False, num_workers=4)

train_anom_loader = data.DataLoader(val_anom_dataset, batch_size=64, shuffle=False,
   drop_last=False, num_workers=4)

To understand the dataset a little better, we can plot below a few sets from the test dataset. Each row shows a different input set, where the first 9 are from the same class.

[32]:
def visualize_exmp(indices, orig_dataset):
    images = [orig_dataset[idx][0] for idx in indices.reshape(-1)]

    images = torch.stack(images, dim=0)
    images = images * TORCH_DATA_STD + TORCH_DATA_MEANS

    img_grid = torchvision.utils.make_grid(images, nrow=SET_SIZE, normalize=True, pad_→value=0.5, padding=16)
    img_grid = img_grid.permute(1, 2, 0)

    plt.figure(figsize=(12,8))
    plt.title("Anomaly examples on CIFAR100")
    plt.imshow(img_grid)
    plt.axis('off')
    plt.show()
    plt.close()

    _, indices, _ = next(iter(test_anom_loader))
    visualize_exmp(indices[:4], test_set)
We can already see that for some sets the task might be easier than for others. Difficulties can especially arise if the anomaly is in a different, but yet visually similar class (e.g. train vs bus, flour vs worm, etc.).

After having prepared the data, we can look closer at the model. Here, we have a classification of the whole set. For the prediction to be permutation-equivariant, we will output one logit for each image. Over these logits, we apply a softmax and train the anomaly image to have the highest score/probability. This is a bit different than a standard classification layer as the softmax is applied over images, not over output classes in the classical sense. However, if we swap two images in their position, we effectively swap their position in the output softmax. Hence, the prediction is equivariant with respect to the input. We implement this idea below in the subclass of the Transformer Lightning module.

```python
[33]: class AnomalyPredictor(TransformerPredictor):

    def _calculate_loss(self, batch, mode="train"):
        img_sets, _, labels = batch
        preds = self.forward(img_sets, add_positional_encoding=False) # No positional encodings as it is a set, not a sequence!
        preds = preds.squeeze(dim=-1) # Shape: [Batch_size, set_size]
        loss = F.cross_entropy(preds, labels) # Softmax/CE over set dimension
        acc = (preds.argmax(dim=-1) == labels).float().mean()
        self.log("%s_loss" % mode, loss)
        self.log("%s_acc" % mode, acc, on_step=False, on_epoch=True)
        return loss, acc

    def training_step(self, batch, batch_idx):
        loss, _ = self._calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="val")

    def test_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="test")
```

Finally, we write our train function below. It has the exact same structure as the reverse task one, hence not much of an explanation is needed here.
Let’s finally train our model. We will use 4 layers with 4 attention heads each. The hidden dimensionality of the model is 256, and we use a dropout of 0.1 throughout the model for good regularization. Note that we also apply the dropout on the input features, as this makes the model more robust against image noise and generalizes better. Again, we use warmup to slowly start our model training.

```python
anomaly_model, anomaly_result = train_anomaly(input_dim=train_anom_dataset.img_feats.shape[-1],
                                              model_dim=256,
                                              num_heads=4,
                                              num_classes=1,
                                              num_layers=4,
                                              dropout=0.1,
                                              input_dropout=0.1,
                                              lr=5e-4,
                                              warmup=100)
```

GPU available: True, used: True
WARNING: Logging before flag parsing goes to stderr.
I1109 10:43:31.036801 139648634296128 distributed.py:49] GPU available: True, used: True
TPU available: False, using: 0 TPU cores
We can print the achieved accuracy below.

```python
[36]: print("Train accuracy: \$%.2f\%" % (100.0*anomaly_result["train_acc"]))
print("Val accuracy: \$%.2f\%" % (100.0*anomaly_result["val_acc"]))
print("Test accuracy: \$%.2f\%" % (100.0*anomaly_result["test_acc"]))
```

Train accuracy: 97.77%
Val accuracy: 94.38%
Test accuracy: 94.30%

With ~94% validation and test accuracy, the model generalizes quite well. It should be noted that you might see slightly different scores depending on what computer/device you are running this notebook. This is because despite setting the seed before generating the test dataset, it is not the same across platforms and numpy versions. Nevertheless, we can conclude that the model performs quite well and can solve the task for most sets. Before trying to interpret the model, let’s verify that our model is permutation-equivariant, and assigns the same predictions for different permutations of the input set. For this, we sample a batch from the test set and run it through the model to obtain the probabilities.

```python
[37]: inp_data, indices, labels = next(iter(test_anom_loader))
inp_data = inp_data.to(device)

anomaly_model.eval()

with torch.no_grad():
    preds = anomaly_model.forward(inp_data, add_positional_encoding=False)
    preds = F.softmax(preds.squeeze(dim=-1), dim=-1)

    # Permut input data
    perm = np.random.permutation(inp_data.shape[1])
    perm_inp_data = inp_data[:,perm]
    perm_preds = anomaly_model.forward(perm_inp_data, add_positional_encoding=False)
    perm_preds = F.softmax(perm_preds.squeeze(dim=-1), dim=-1)

    assert (preds[:,perm] - perm_preds).abs().max() < 1e-5, "Predictions are not permutation equivariant"
```

(continues on next page)
print("Preds
", preds[0,permut].cpu().numpy())
print("Permuted preds
", perm_preds[0].cpu().numpy())

Preds
[5.4543594e-05 1.4208173e-04 6.6922468e-05 7.6413504e-05 7.7112330e-05
Permuted preds
[5.4543532e-05 1.4208158e-04 6.6922395e-05 7.6413417e-05 7.7112243e-05
8.7848362e-05 6.6820678e-05 9.9929142e-01 7.3219751e-05 6.3545544e-05]

You can see that the predictions are almost exactly the same, and only differ because of slight numerical differences inside the network operation.

To interpret the model a little more, we can plot the attention maps inside the model. This will give us an idea of what information the model is sharing/communicating between images, and what each head might represent. First, we need to extract the attention maps for the test batch above, and determine the discrete predictions for simplicity.

[38]: attention_maps = anomaly_model.get_attention_maps(inp_data, add_positional_encoding=False)
predictions = preds.argmax(dim=-1)

Below we write a plot function which plots the images in the input set, the prediction of the model, and the attention maps of the different heads on layers of the transformer. Feel free to explore the attention maps for different input examples as well.

[39]: def visualize_prediction(idx):
    visualize_exmp(indices[idx:idx+1], test_set)
    print("Prediction:", predictions[idx].item())
    plot_attention_maps(input_data=None, attn_maps=attention_maps, idx=idx)

visualize_prediction(0)

Anomaly examples on CIFAR100

Prediction: 9
Depending on the random seed, you might see a slightly different input set. For the version on the website, we compare 9 tree images with a volcano. We see that multiple heads, for instance, Layer 2 Head 1, Layer 2 Head 3, and Layer 3 Head 1 focus on the last image. Additionally, the heads in Layer 4 all seem to ignore the last image and assign a very low attention probability to it. This shows that the model has indeed recognized that the image doesn’t fit the setting, and hence predicted it to be the anomaly. Layer 3 Head 2-4 seems to take a slightly weighted average of all images. That might indicate that the model extracts the “average” information of all images, to compare it to the image features itself.

Let’s try to find where the model actually makes a mistake. We can do this by identifying the sets where the model predicts something else than 9, as in the dataset, we ensured that the anomaly is always at the last position in the set.
As our model achieves ~94% accuracy, we only have very little number of mistakes in a batch of 64 sets. Still, let’s visualize one of them, for example the last one:

```python
visualize_prediction(mistakes[-1])
print("Probabilities:")
for i, p in enumerate(preds[mistakes[-1]].cpu().numpy()):
    print("Image \$i: \$4.2f\$\% \(i, 100.0*p)\")
```

Anomaly examples on CIFAR100

Prediction: 2
In this example, the model confuses a palm tree with a building, giving a probability of ~90% to image 2, and 8% to
the actual anomaly. However, the difficulty here is that the picture of the building has been taken at a similar angle as
the palms. Meanwhile, image 2 shows a rather unusual palm with a different color palette, which is why the model
fails here. Nevertheless, in general, the model performs quite well.

5.8.3 Conclusion

In this tutorial, we took a closer look at the Multi-Head Attention layer which uses a scaled dot product between
queries and keys to find correlations and similarities between input elements. The Transformer architecture is based
on the Multi-Head Attention layer and applies multiple of them in a ResNet-like block. The Transformer is a very
important, recent architecture that can be applied to many tasks and datasets. Although it is best known for its success
in NLP, there is so much more to it. We have seen its application on sequence-to-sequence tasks and set anomaly
detection. Its property of being permutation-equivariant if we do not provide any positional encodings, allows it to
generalize to many settings. Hence, it is important to know the architecture, but also its possible issues such as the
gradient problem during the first iterations solved by learning rate warm-up. If you are interested in continuing with
the study of the Transformer architecture, please have a look at the blog posts listed at the beginning of the tutorial
notebook.

5.9 Tutorial 7: Graph Neural Networks

Filled notebook:

Pre-trained models:

In this tutorial, we will discuss the application of neural networks on graphs. Graph Neural Networks (GNNs) have
recently gained increasing popularity in both applications and research, including domains such as social networks,
knowledge graphs, recommender systems, and bioinformatics. While the theory and math behind GNNs might first
seem complicated, the implementation of those models is quite simple and helps in understanding the methodology.
Therefore, we will discuss the implementation of basic network layers of a GNN, namely graph convolutions, and
attention layers. Finally, we will apply a GNN on a node-level, edge-level, and graph-level tasks.

Below, we will start by importing our standard libraries. We will use PyTorch Lightning as already done in Tutorial 5
and 6.

```python
## Standard libraries
import os
import json
import math
import numpy as np
import time

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

## Progress bar
```
from tqdm.notebook import tqdm

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

# Torchvision
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms

# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError:
    # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install pytorch-lightning==1.0.3
import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial7"

# Setting the seed
pl.seed_everything(42)
# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

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

We also have a few pre-trained models we download below.

```python
import urllib.request
from urllib.error import HTTPError

# Github URL where saved models are stored for this tutorial

# Files to download
pretrained_files = ["NodeLevelMLP.ckpt", "NodeLevelGNN.ckpt", "GraphLevelGraphConv.ckpt"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "" in file_name:
        os.makedirs(file_path.rsplit("/",1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        # Download the file
        try:
            urllib.request.urlretrieve(base_url + file_name, file_path)
        except HTTPError:
            print(f"Failed to download {file_name}")
```

(continues on next page)
5.9.1 Graph Neural Networks

Graph representation

Before starting the discussion of specific neural network operations on graphs, we should consider how to represent a graph. Mathematically, a graph $\mathcal{G}$ is defined as a tuple of a set of nodes/vertices $V$, and a set of edges/links $E$: $\mathcal{G} = (V, E)$. Each edge is a pair of two vertices, and represents a connection between them. For instance, let’s look at the following graph:

The vertices are $V = \{1, 2, 3, 4\}$, and edges $E = \{(1, 2), (2, 3), (2, 4), (3, 4)\}$. Note that for simplicity, we assume the graph to be undirected and hence don’t add mirrored pairs like $(2, 1)$. In application, vertices and edge can often have specific attributes, and edges can even be directed. The question is how we could represent this diversity in an efficient way for matrix operations. Usually, for the edges, we decide between two variants: an adjacency matrix, or a list of paired vertex indices.

The adjacency matrix $A$ is a square matrix whose elements indicate whether pairs of vertices are adjacent, i.e. connected, or not. In the simplest case, $A_{ij}$ is 1 if there is a connection from node $i$ to $j$, and otherwise 0. If we have edge attributes or different categories of edges in a graph, this information can be added to the matrix as well. For an undirected graph, keep in mind that $A$ is a symmetric matrix ($A_{ij} = A_{ji}$). For the example graph above, we have the following adjacency matrix:

$$
A = \begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0
\end{bmatrix}
$$

While expressing a graph as a list of edges is more efficient in terms of memory and (possibly) computation, using an adjacency matrix is more intuitive and simpler to implement. In our implementations below, we will rely on the adjacency matrix to keep the code simple. However, common libraries use edge lists, which we will discuss later more. Alternatively, we could also use the list of edges to define a sparse adjacency matrix with which we can work as if it was a dense matrix, but allows more memory-efficient operations. PyTorch supports this with the sub-package `torch.sparse` (documentation) which is however still in a beta-stage (API might change in future).
Graph Convolutions

Graph Convolutional Networks have been introduced by Kipf et al. in 2016 at the University of Amsterdam. He also wrote a great blog post about this topic, which is recommended if you want to read about GCNs from a different perspective. GCNs are similar to convolutions in images in the sense that the “filter” parameters are typically shared over all locations in the graph. At the same time, GCNs rely on message passing methods, which means that vertices exchange information with the neighbors, and send “messages” to each other. Before looking at the math, we can try to visually understand how GCNs work. The first step is that each node creates a feature vector that represents the message it wants to send to all its neighbors. In the second step, the messages are sent to the neighbors, so that a node receives one message per adjacent node. Below we have visualized the two steps for our example graph.

If we want to formulate that in more mathematical terms, we need to first decide how to combine all the messages a node receives. As the number of messages vary across nodes, we need an operation that works for any number. Hence, the usual way to go is to sum or take the mean. Given the previous features of nodes $H^{(l)}$, the GCN layer is defined as follows:

$$ H^{(l+1)} = \sigma \left( \hat{D}^{-1/2} \hat{A} \hat{D}^{-1/2} H^{(l)} W^{(l)} \right) $$

$W^{(l)}$ is the weight parameters with which we transform the input features into messages ($H^{(l)} W^{(l)}$). To the adjacency matrix $A$ we add the identity matrix so that each node sends its own message also to itself: $\hat{A} = A + I$. Finally, to take the average instead of summing, we calculate the matrix $\hat{D}$ which is a diagonal matrix with $D_{ii}$ denoting the number of neighbors node $i$ has. $\sigma$ represents an arbitrary activation function, and not necessarily the sigmoid (usually a ReLU-based activation function is used in GNNs).

When implementing the GCN layer in PyTorch, we can take advantage of the flexible operations on tensors. Instead of defining a matrix $\hat{D}$, we can simply divide the summed messages by the number of neighbors afterward. Additionally, we replace the weight matrix with a linear layer, which additionally allows us to add a bias. Written as a PyTorch module, the GCN layer is defined as follows:

```python
import torch.nn as nn

class GCNLayer(nn.Module):
    def __init__(self, c_in, c_out):
        super().__init__()
        self.projection = nn.Linear(c_in, c_out)

    def forward(self, node_feats, adj_matrix):
        # Num neighbours = number of incoming edges
        num_neighbours = adj_matrix.sum(dim=-1, keepdim=True)
        node_feats = self.projection(node_feats)
        node_feats = torch.bmm(adj_matrix, node_feats)
        node_feats = node_feats / num_neighbours
        return node_feats
```

To further understand the GCN layer, we can apply it to our example graph above. First, let’s specify some node features and the adjacency matrix with added self-connections:
Node features:
tensor([[0., 1.],
         [2., 3.],
         [4., 5.],
         [6., 7.]]))

Adjacency matrix:
tensor([[1., 1., 0., 0.],
         [1., 1., 1., 1.],
         [0., 1., 1., 1.],
         [0., 1., 1., 1.]]))

Next, let’s apply a GCN layer to it. For simplicity, we initialize the linear weight matrix as an identity matrix so that
the input features are equal to the messages. This makes it easier for us to verify the message passing operation.

```
[5]: layer = GCNLayer(c_in=2, c_out=2)
layer.projection.weight.data = torch.Tensor([[1., 0.],
                                                 [0., 1.])
layer.projection.bias.data = torch.Tensor([0., 0.])

with torch.no_grad():
    out_feats = layer(node_feats, adj_matrix)
```

Adjacency matrix tensor([[1., 1., 0., 0.],
                          [1., 1., 1., 1.],
                          [0., 1., 1., 1.],
                          [0., 1., 1., 1.]]))

Input features tensor([[0., 1.],
                        [2., 3.],
                        [4., 5.],
                        [6., 7.]]))

Output features tensor([[1.2., 2.],
                         [3.4., 5.],
                         [4.5., 5.]]))

As we can see, the first node’s output values are the average of itself and the second node. Similarly, we can verify all
other nodes. However, in a GNN, we would also want to allow feature exchange between nodes beyond its neighbors.
This can be achieved by applying multiple GCN layers, which gives us the final layout of a GNN. The GNN can be
build up by a sequence of GCN layers and non-linearities such as ReLU. For a visualization, see below (figure credit
- Thomas Kipf, 2016).
However, one issue we can see from looking at the example above is that the output features for nodes 3 and 4 are the same because they have the same adjacent nodes (including itself). Therefore, GCN layers can make the network forget node-specific information if we just take a mean over all messages. Multiple possible improvements have been proposed. While the simplest option might be using residual connections, the more common approach is to either weigh the self-connections higher or define a separate weight matrix for the self-connections. Alternatively, we can re-visit a concept from the last tutorial: attention.

Graph Attention

If you remember from the last tutorial, attention describes a weighted average of multiple elements with the weights dynamically computed based on an input query and elements’ keys (if you haven’t read Tutorial 6 yet, it is recommended to at least go through the very first section called What is Attention?). This concept can be similarly applied to graphs, one of such is the Graph Attention Network (called GAT, proposed by Velickovic et al., 2017). Similarly to the GCN, the graph attention layer creates a message for each node using a linear layer/weight matrix. For the attention part, it uses the message from the node itself as a query, and the messages to average as both keys and values (note that this also includes the message to itself). The score function $f_{\text{attn}}$ is implemented as a one-layer MLP which maps the query and key to a single value. The MLP looks as follows (figure credit - Velickovic et al.):

$$
\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(a \|Wh_i\|Wh_j))}{\sum_{k \in \mathcal{N}_i} \exp(\text{LeakyReLU}(a \|Wh_i\|Wh_k))}
$$

The operator $\|$ represents the concatenation, and $\mathcal{N}_i$ the indices of the neighbors of node $i$. Note that in contrast to usual practice, we apply a non-linearity (here LeakyReLU) before the softmax over elements. Although it seems like a minor change at first, it is crucial for the attention to depend on the original input. Specifically, let’s remove the
non-linearity for a second, and try to simplify the expression:

\[ \alpha_{ij} = \frac{\exp (a \cdot [W_h_i || W_h_j])}{\sum_{k \in N_i} \exp (a \cdot [W_h_i || W_h_k])} \]

\[ = \frac{\exp (a_{z:d/2} W_{h_i} + a_{z:d/2} W_{h_j})}{\sum_{k \in N_i} \exp (a_{z:d/2} W_{h_i} + a_{z:d/2} W_{h_k})} \]

\[ = \frac{\exp (a_{z:d/2} W_{h_i}) \cdot \exp (a_{z:d/2} W_{h_j})}{\sum_{k \in N_i} \exp (a_{z:d/2} W_{h_i}) \cdot \exp (a_{z:d/2} W_{h_k})} \]

\[ = \frac{\exp (a_{z:d/2} W_{h_j})}{\sum_{k \in N_i} \exp (a_{z:d/2} W_{h_k})} \]

We can see that without the non-linearity, the attention term with \( h_i \) actually cancels itself out, resulting in the attention being independent of the node itself. Hence, we would have the same issue as the GCN of creating the same output features for nodes with the same neighbors. This is why the LeakyReLU is crucial and adds some dependency on \( h_i \) to the attention.

Once we obtain all attention factors, we can calculate the output features for each node by performing the weighted average:

\[ h_i' = \sigma \left( \sum_{j \in N_i} \alpha_{ij} W_{h_j} \right) \]

\( \sigma \) is yet another non-linearity, as in the GCN layer. Visually, we can represent the full message passing in an attention layer as follows (figure credit - Velickovic et al.):

![Attention Layer Diagram](image)

To increase the expressiveness of the graph attention network, Velickovic et al. proposed to extend it to multiple heads similar to the Multi-Head Attention block in Transformers. This results in \( N \) attention layers being applied in parallel. In the image above, it is visualized as three different colors of arrows (green, blue, and purple) that are afterward concatenated. The average is only applied for the very final prediction layer in a network.

After having discussed the graph attention layer in detail, we can implement it below:

[6]:
```python
class GATLayer(nn.Module):
    def __init__(self, c_in, c_out, num_heads=1, concat_heads=True, alpha=0.2):
        (continues on next page)```
```python
***

Inputs:
- `c_in` - Dimensionality of input features
- `c_out` - Dimensionality of output features
- `num_heads` - Number of heads, i.e. attention mechanisms to apply in parallel. The output features are equally split up over the heads if `concat_heads=True`.
- `concat_heads` - If True, the output of the different heads is concatenated instead of averaged.
- `alpha` - Negative slope of the LeakyReLU activation.

super().__init__()
self.num_heads = num_heads
self.concat_heads = concat_heads
if self.concat_heads:
    assert c_out % num_heads == 0, "Number of output features must be a multiple of the count of heads."
    c_out = c_out // num_heads

# Sub-modules and parameters needed in the layer
self.projection = nn.Linear(c_in, c_out * num_heads)
self.a = nn.Parameter(torch.Tensor(num_heads, 2 * c_out))  # One per head
self.leakyrelu = nn.LeakyReLU(alpha)

# Initialization from the original implementation
nn.init.xavier_uniform_(self.projection.weight.data, gain=1.414)
nn.init.xavier_uniform_(self.a.data, gain=1.414)

def forward(self, node_feats, adj_matrix, print_attn_probs=False):
    ***
    Inputs:
    - `node_feats` - Input features of the node. Shape: [batch_size, c_in]
    - `adj_matrix` - Adjacency matrix including self-connections. Shape: [batch_size, num_nodes, num_nodes]
    - `print_attn_probs` - If True, the attention weights are printed during the forward pass (for debugging purposes)
    ***
    batch_size, num_nodes = node_feats.size(0), node_feats.size(1)

    # Apply linear layer and sort nodes by head
    node_feats = self.projection(node_feats)
    node_feats = node_feats.view(batch_size, num_nodes, self.num_heads, -1)

    # We need to calculate the attention logits for every edge in the adjacency matrix
    # => Create a tensor of [W*h_i||W*h_j] with i and j being the indices of all edges
    edges = adj_matrix.nonzero(as_tuple=False)  # Returns indices where the adjacency matrix is not 0 => edges
    node_feats_flattened = node_feats.view(batch_size * num_nodes, self.num_heads, -1)
    edge_indices_row = edges[:, 0] * batch_size + edges[:, 1]
    edge_indices_col = edges[:, 0] * batch_size + edges[:, 2]
    a_input = torch.cat([torch.index_select(input=node_feats_flattened, index=edge_indices_row, dim=0),
                         torch.index_select(input=node_feats_flattened, index=edge_indices_col, dim=0)], dim=2)
```

(continues on previous page)
Again, we can apply the graph attention layer on our example graph above to understand the dynamics better. As before, the input layer is initialized as an identity matrix, but we set a to be a vector of arbitrary numbers to obtain different attention values. We use two heads to show the parallel, independent attention mechanisms working in the layer.

```
[7]: layer = GATLayer(2, 2, num_heads=2)
layer.projection.weight.data = torch.Tensor([[1., 0.], [0., 1.]])
layer.projection.bias.data = torch.Tensor([0., 0.])
layer.a.data = torch.Tensor([[-0.2, 0.3], [0.1, -0.1]])
with torch.no_grad():
    out_feats = layer(node_feats, adj_matrix, print_attn_probs=True)

print("Adjacency matrix", adj_matrix)
print("Input features", node_feats)
print("Output features", out_feats)
```

**Attention probs**
```
tensor([[0.3543, 0.6457, 0.0000, 0.0000],
        [0.1096, 0.1450, 0.2642, 0.4813],
        [0.0000, 0.1858, 0.2885, 0.5257],
        [0.0000, 0.2391, 0.2696, 0.4913]],
       [[0.5100, 0.4900, 0.0000, 0.0000],
        [0.2975, 0.2436, 0.2340, 0.2249],
        [0.0000, 0.3838, 0.3142, 0.3019],
        [0.0000, 0.4018, 0.3289, 0.2693]]))
```

**Adjacency matrix**
```
tensor([[1., 1., 0., 0.],
         [1., 1., 0., 0.],
         [0., 0., 1., 1.],
         [0., 0., 1., 1.]])
```
We recommend that you try to calculate the attention matrix at least for one head and one node for yourself. The entries are 0 where there does not exist an edge between $i$ and $j$. For the others, we see a diverse set of attention probabilities. Moreover, the output features of node 3 and 4 are now different although they have the same neighbors.

### 5.9.2 PyTorch Geometric

We had mentioned before that implementing graph networks with adjacency matrix is simple and straight-forward but can be computationally expensive for large graphs. Many real-world graphs can reach over 200k nodes, for which adjacency matrix-based implementations fail. There are a lot of optimizations possible when implementing GNNs, and luckily, there exist packages that provide such layers. The most popular packages for PyTorch are PyTorch Geometric and the Deep Graph Library (the latter being actually framework agnostic). Which one to use depends on the project you are planning to do and personal taste. In this tutorial, we will look at PyTorch Geometric as part of the PyTorch family. Similar to PyTorch Lightning, PyTorch Geometric is not installed by default on GoogleColab (and actually also not in our `dl2020` environment due to many dependencies that would be unnecessary for the practicals). Hence, let’s import and/or install it below:

```python
# torch geometric
try:
    import torch_geometric
except ModuleNotFoundError:
    # You might need to install those packages with specific CUDA+PyTorch version.
    # The following ones below have been picked for Colab (Nov 2020).
    # See https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html for details
    !pip install torch-geometric
import torch_geometric
import torch_geometric.nn as geom_nn
import torch_geometric.data as geom_data
```

PyTorch Geometric provides us a set of common graph layers, including the GCN and GAT layer we implemented above. Additionally, similar to PyTorch’s torchvision, it provides the common graph datasets and transformations on those to simplify training. Compared to our implementation above, PyTorch Geometric uses a list of index pairs to represent the edges. The details of this library will be explored further in our experiments.
In our tasks below, we want to allow us to pick from a multitude of graph layers. Thus, we define again below a
dictionary to access those using a string:

```
[9]: gnn_layer_by_name = {
    "GCN": geom_nn.GCNConv,
    "GAT": geom_nn.GATConv,
    "GraphConv": geom_nn.GraphConv
}
```

Additionally to GCN and GAT, we added the layer `geom_nn.GraphConv` (documentation). GraphConv is a GCN
with a separate weight matrix for the self-connections. Mathematically, this would be:

\[
x^{(l+1)}_i = W^{(l+1)}_1 x^{(l)}_i + W^{(l+1)}_2 \sum_{j \in N_i} x^{(l)}_j
\]

In this formula, the neighbor’s messages are added instead of averaged. However, PyTorch Geometric provides the
argument `aggr` to switch between summing, averaging, and max pooling.

### 5.9.3 Experiments on graph structures

Tasks on graph-structured data can be grouped into three groups: node-level, edge-level and graph-level. The different
levels describe on which level we want to perform classification/regression. We will discuss all three types in more
detail below.

**Node-level tasks: Semi-supervised node classification**

Node-level tasks have the goal to classify nodes in a graph. Usually, we have given a single, large graph with >1000
nodes of which a certain amount of nodes are labeled. We learn to classify those labeled examples during training and
try to generalize to the unlabeled nodes.

A popular example that we will use in this tutorial is the Cora dataset, a citation network among papers. The Cora
consists of 2708 scientific publications with links between each other representing the citation of one paper by another.
The task is to classify each publication into one of seven classes. Each publication is represented by a bag-of-words
vector. This means that we have a vector of 1433 elements for each publication, where a 1 at feature \(i\) indicates that
the \(i\)-th word of a pre-defined dictionary is in the article. Binary bag-of-words representations are commonly used
when we need very simple encodings, and already have an intuition of what words to expect in a network. There exist
much better approaches, but we will leave this to the NLP courses to discuss.

We will load the dataset below:

```
[10]: cora_dataset = torch_geometric.datasets.Planetoid(root=DATASET_PATH, name="Cora")
```

Let’s look at how PyTorch Geometric represents the graph data. Note that although we have a single graph, PyTorch
Geometric returns a dataset for compatibility to other datasets.

```
[11]: cora_dataset[0]
```

```
Data(edge_index=[2, 10556], test_mask=[2708], train_mask=[2708], val_mask=[2708], x=[2708, 1433], y=[2708])
```

The graph is represented by a `Data` object (documentation) which we can access as a standard Python namespace.
The edge index tensor is the list of edges in the graph and contains the mirrored version of each edge for undirected
graphs. The `train_mask`, `val_mask`, and `test_mask` are boolean masks that indicate which nodes we should use for training, validation, and testing. The `x` tensor is the feature tensor of our 2708 publications, and `y` the labels
for all nodes.
After having seen the data, we can implement a simple graph neural network. The GNN applies a sequence of graph layers (GCN, GAT, or GraphConv), ReLU as activation function, and dropout for regularization. See below for the specific implementation.

```python
[12]: class GNNModel(nn.Module):

    def __init__(self, c_in, c_hidden, c_out, num_layers=2, layer_name="GCN", dp_rate=0.1, **kwargs):
        """
        Inputs:
            c_in - Dimension of input features
            c_hidden - Dimension of hidden features
            c_out - Dimension of the output features. Usually number of classes in classification
            num_layers - Number of "hidden" graph layers
            layer_name - String of the graph layer to use
            dp_rate - Dropout rate to apply throughout the network
            kwargs - Additional arguments for the graph layer (e.g. number of heads for GAT)
        """
        super().__init__()
        gnn_layer = gnn_layer_by_name[layer_name]
        layers = []
        in_channels, out_channels = c_in, c_hidden
        for l_idx in range(num_layers-1):
            layers += [
                gnn_layer(in_channels=in_channels,
                        out_channels=out_channels,
                        **kwargs),
                nn.ReLU(inplace=True),
                nn.Dropout(dp_rate)
            ]
        in_channels = c_hidden
        layers += [gnn_layer(in_channels=in_channels,
                        out_channels=c_out,
                        **kwargs)]
        self.layers = nn.ModuleList(layers)

    def forward(self, x, edge_index):
        """
        Inputs:
            x - Input features per node
            edge_index - List of vertex index pairs representing the edges in the graph (PyTorch geometric notation)
        """
        for l in self.layers:
            if isinstance(l, geom.nn.MessagePassing):
                x = l(x, edge_index)
            else:
                x = l(x)
        return x
```

Good practice in node-level tasks is to create an MLP baseline that is applied to each node independently. This way
we can verify whether adding the graph information to the model indeed improves the prediction, or not. It might also
be that the features per node are already expressive enough to clearly point towards a specific class. To check this, we
implement a simple MLP below.

```python
[13]: class MLPModel(nn.Module):
    def __init__(self, c_in, c_hidden, c_out, num_layers=2, dp_rate=0.1):
        """
        Inputs:
        c_in - Dimension of input features
        c_hidden - Dimension of hidden features
        c_out - Dimension of the output features. Usually number of classes in
        Classification
        num_layers - Number of hidden layers
        dp_rate - Dropout rate to apply throughout the network
        """
        super().__init__()
        layers = []
        in_channels, out_channels = c_in, c_hidden
        for l_idx in range(num_layers-1):
            layers += [
                nn.Linear(in_channels, out_channels),
                nn.ReLU(inplace=True),
                nn.Dropout(dp_rate)
            ]
            in_channels = c_hidden
        layers += [nn.Linear(in_channels, c_out)]
        self.layers = nn.Sequential(*layers)

    def forward(self, x, *args, **kwargs):
        """
        Inputs:
        x - Input features per node
        """
        return self.layers(x)
```

Finally, we can merge the models into a PyTorch Lightning module which handles the training, validation, and testing
for us.

```python
[14]: class NodeLevelGNN(pl.LightningModule):
    def __init__(self, model_name, **model_kwargs):
        super().__init__()
        # Saving hyperparameters
        self.save_hyperparameters()

        if model_name == "MLP":
            self.model = MLPModel(**model_kwargs)
        else:
            self.model = GNNModel(**model_kwargs)
        self.loss_module = nn.CrossEntropyLoss()

    def forward(self, data, mode="train"):
        x, edge_index = data.x, data.edge_index
        x = self.model(x, edge_index)

        # Only calculate the loss on the nodes corresponding to the mask
```

(continues on next page)
if mode == "train":
    mask = data.train_mask
elif mode == "val":
    mask = data.val_mask
elif mode == "test":
    mask = data.test_mask
else:
    assert False, "Unknown forward mode: %s" % mode

loss = self.loss_module(x[mask], data.y[mask])
acc = (x[mask].argmax(dim=-1) == data.y[mask]).sum().float() / mask.sum()
return loss, acc

def configure_optimizers(self):
    # We use SGD here, but Adam works as well
    optimizer = optim.SGD(self.parameters(), lr=0.1, momentum=0.9, weight_decay=2e-3)
    return optimizer

def training_step(self, batch, batch_idx):
    loss, acc = self.forward(batch, mode="train")
    self.log('train_loss', loss)
    self.log('train_acc', acc)
    return loss

def validation_step(self, batch, batch_idx):
    _, acc = self.forward(batch, mode="val")
    self.log('val_acc', acc)

def test_step(self, batch, batch_idx):
    _, acc = self.forward(batch, mode="test")
    self.log('test_acc', acc)

Additionally to the Lightning module, we define a training function below. As we have a single graph, we use a batch size of 1 for the data loader and share the same data loader for the train, validation, and test set (the mask is picked inside the Lightning module). Besides, we set the argument `progress_bar_refresh_rate` to zero as it usually shows the progress per epoch, but an epoch only consists of a single step. The rest of the code is very similar to what we have seen in Tutorial 5 and 6 already.

[15]: def train_node_classifier(model_name, dataset, **model_kwargs):
    pl.seed_everything(42)
    node_data_loader = geom_data.DataLoader(dataset, batch_size=1)
    root_dir = os.path.join(CHECKPOINT_PATH, "NodeLevel" + model_name)
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir, exist_ok=True, mode="max", monitor="val_acc"),
    gpus=1 if str(device).startswith("cuda") else 0,
    max_epochs=200,
    progress_bar_refresh_rate=0) # 0 because epoch size is 1

(continues on next page)
Finally, we can train our models. First, let’s train the simple MLP:

```python
# Small function for printing the test scores
def print_results(result_dict):
    if "train" in result_dict:
        print("Train accuracy: {:.2f}%" .format(100.0*result_dict["train"]))
    if "val" in result_dict:
        print("Val accuracy: {:.2f}%" .format(100.0*result_dict["val"]))
    print("Test accuracy: {:.2f}%" .format(100.0*result_dict["test"]))

node_mlp_model, node_mlp_result = train_node_classifier(model_name="MLP",
                                                        dataset=cora_dataset,
                                                        c_hidden=16,
                                                        num_layers=2,
                                                        dp_rate=0.1)

print_results(node_mlp_result)
```

GPU available: True, used: True
WARNING: Logging before flag parsing goes to stderr.
I1113 19:12:52.401648 139969460983616 distributed.py:49] GPU available: True, used: True
TPU available: False, using: 0 TPU cores
I1113 19:12:52.403518 139969460983616 distributed.py:49] TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
I1113 19:12:52.404974 139969460983616 accelerator_connector.py:385] LOCAL_RANK: 0 -
"CUDA_VISIBLE_DEVICES: [0]"
Although the MLP can overfit on the training dataset because of the high-dimensional input features, it does not perform too well on the test set. Let’s see if we can beat this score with our graph networks:

```
[18]: node_gnn_model, node_gnn_result = train_node_classifier(model_name="GNN",
                     layer_name="GCN",
                     dataset=cora_dataset,
                     c_hidden=16,
                     num_layers=2,
                     dp_rate=0.1)
print_results(node_gnn_result)
```

As we would have hoped for, the GNN model outperforms the MLP by quite a margin. This shows that using the graph information indeed improves our predictions and lets us generalize better.

The hyperparameters in the model have been chosen to create a relatively small network. This is because the first layer with an input dimension of 1433 can be relatively expensive to perform for large graphs. In general, GNNs can become relatively expensive for very big graphs. This is why such GNNs either have a small hidden size or use a special batching strategy where we sample a connected subgraph of the big, original graph.

**Edge-level tasks: Link prediction**

In some applications, we might have to predict on an edge-level instead of node-level. The most common edge-level task in GNN is link prediction. Link prediction means that given a graph, we want to predict whether there will be/should be an edge between two nodes or not. For example, in a social network, this is used by Facebook and co to propose new friends to you. Again, graph level information can be crucial to perform this task. The output prediction is usually done by performing a similarity metric on the pair of node features, which should be 1 if there should be a link, and otherwise close to 0. To keep the tutorial short, we will not implement this task ourselves. Nevertheless, there are many good resources out there if you are interested in looking closer at this task. Tutorials and papers for this topic include:

- PyTorch Geometric example
• Graph Neural Networks: A Review of Methods and Applications, Zhou et al. 2019
• Link Prediction Based on Graph Neural Networks, Zhang and Chen, 2018.

Graph-level tasks: Graph classification

Finally, in this part of the tutorial, we will have a closer look at how to apply GNNs to the task of graph classification. The goal is to classify an entire graph instead of single nodes or edges. Therefore, we are also given a dataset of multiple graphs that we need to classify based on some structural graph properties. The most common task for graph classification is molecular property prediction, in which molecules are represented as graphs. Each atom is linked to a node, and edges in the graph are the bonds between atoms. For example, look at the figure below.

On the left, we have an arbitrary, small molecule with different atoms, whereas the right part of the image shows the graph representation. The atom types are abstracted as node features (e.g. a one-hot vector), and the different bond types are used as edge features. For simplicity, we will neglect the edge attributes in this tutorial, but you can include by using methods like the Relational Graph Convolution that uses a different weight matrix for each edge type.

The dataset we will use below is called the MUTAG dataset. It is a common small benchmark for graph classification algorithms, and contain 188 graphs with 18 nodes and 20 edges on average for each graph. The graph nodes have 7 different labels/atom types, and the binary graph labels represent “their mutagenic effect on a specific gram negative bacterium” (the specific meaning of the labels are not too important here). The dataset is part of a large collection of different graph classification datasets, known as the TUDatasets, which is directly accessible via torch_geometric.datasets.TUDataset (documentation) in PyTorch Geometric. We can load the dataset below.

```python
[19]: tu_dataset = torch_geometric.datasets.TUDataset(root=DATASET_PATH, name="MUTAG")
```

Let’s look at some statistics for the dataset:

```python
[20]: print("Data object:", tu_dataset.data) print("Length:", len(tu_dataset)) print("Average label: %.2f" % (tu_dataset.data.y.float().mean().item()))
```

```
Data object: Data(edge_attr=[7442, 4], edge_index=[2, 7442], x=[3371, 7], y=[188])
Length: 188
Average label: 0.66
```

The first line shows how the dataset stores different graphs. The nodes, edges, and labels of each graph are concatenated to one tensor, and the dataset stores the indices where to split the tensors correspondingly. The length of the dataset is the number of graphs we have, and the “average label” denotes the percentage of the graph with label 1. As long as the percentage is in the range of 0.5, we have a relatively balanced dataset. It happens quite often that graph datasets are very imbalanced, hence checking the class balance is always a good thing to do.

Next, we will split our dataset into a training and test part. Note that we do not use a validation set this time because of the small size of the dataset. Therefore, our model might overfit slightly on the validation set due to the noise of the evaluation, but we still get an estimate of the performance on untrained data.

```python
[21]: torch.manual_seed(42) tu_dataset.shuffle() train_dataset = tu_dataset[:150] test_dataset = tu_dataset[150:]
```

When using a data loader, we encounter a problem with batching \( N \) graphs. Each graph in the batch can have a different number of nodes and edges, and hence we would require a lot of padding to obtain a single tensor. Torch geometric uses a different, more efficient approach: we can view the \( N \) graphs in a batch as a single large graph with concatenated node and edge list. As there is no edge between the \( N \) graphs, running GNN layers on the large graph...
gives us the same output as running the GNN on each graph separately. Visually, this batching strategy is visualized below (figure credit - PyTorch Geometric team, tutorial here).

\[ G_1 = (X_1, A_1) \xrightarrow{\text{GNN}} (A_1, X_1) = X'_1 \]

\[ G_2 = (X_2, A_2) \xrightarrow{\text{GNN}} (A_2, X_2) = X'_2 \]

The adjacency matrix is zero for any nodes that come from two different graphs, and otherwise according to the adjacency matrix of the individual graph. Luckily, this strategy is already implemented in torch geometric, and hence we can use the corresponding data loader:

```python
[22]: graph_train_loader = geom_data.DataLoader(train_dataset, batch_size=64, shuffle=True)
graph_val_loader = geom_data.DataLoader(test_dataset, batch_size=64) # Additional loader if you want to change to a larger dataset
graph_test_loader = geom_data.DataLoader(test_dataset, batch_size=64)
```

Let’s load a batch below to see the batching in action:

```python
[23]: batch = next(iter(graph_test_loader))
print("Batch:", batch)
print("Labels:", batch.y[:10])
print("Batch indices:", batch.batch[:40])
```

```
Batch: Batch(batch=[687], edge_attr=[1512, 4], edge_index=[2, 1512], x=[687, 7], y=[38])
Labels: tensor([1, 1, 1, 0, 0, 0, 1, 1, 1, 0])
Batch indices: tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2])
```

We have 38 graphs stacked together for the test dataset. The batch indices, stored in `batch`, show that the first 12 nodes belong to the first graph, the next 22 to the second graph, and so on. These indices are important for performing the final prediction. To perform a prediction over a whole graph, we usually perform a pooling operation over all nodes after running the GNN model. In this case, we will use the average pooling. Hence, we need to know which nodes should be included in which average pool. Using this pooling, we can already create our graph network below. Specifically, we re-use our class `GNNModel` from before, and simply add an average pool and single linear layer for the graph prediction task.

```python
[24]: class GraphGNNModel(nn.Module):
    def __init__(self, c_in, c_hidden, c_out, dp_rate_linear=0.5, **kwargs):
        """
        Inputs:
        c_in - Dimension of input features
        c_hidden - Dimension of hidden features
        c_out - Dimension of output features (usually number of classes)
        dp_rate_linear - Dropout rate before the linear layer (usually much higher than inside the GNN)
        kwargs - Additional arguments for the GNNModel object
        """
        super().__init__()
        self.GNN = GNNModel(c_in=c_in,
```

(continues on next page)
```python
self.head = nn.Sequential(
    nn.Dropout(dp_rate_linear),
    nn.Linear(c_hidden, c_out)
)

def forward(self, x, edge_index, batch_idx):
    """
    Inputs:
    x - Input features per node
    edge_index - List of vertex index pairs representing the edges in the graph (PyTorch geometric notation)
    batch_idx - Index of batch element for each node
    """
    x = self.GNN(x, edge_index)
    x = geom_nn.global_mean_pool(x, batch_idx)  # Average pooling
    x = self.head(x)
    return x
```

Finally, we can create a PyTorch Lightning module to handle the training. It is similar to the modules we have seen before and does nothing surprising in terms of training. As we have a binary classification task, we use the Binary Cross Entropy loss.

```python
class GraphLevelGNN(pl.LightningModule):
    def __init__(self, **model_kwargs):
        super().__init__()
        # Saving hyperparameters
        self.save_hyperparameters()
        self.model = GraphGNNModel(**model_kwargs)
        self.loss_module = nn.BCEWithLogitsLoss() if self.hparams.c_out == 1 else nn.CrossEntropyLoss()

    def forward(self, data, mode="train"):
        x, edge_index, batch_idx = data.x, data.edge_index, data.batch
        x = self.model(x, edge_index, batch_idx)
        x = x.squeeze(dim=-1)
        if self.hparams.c_out == 1:
            preds = (x > 0).float()
            data.y = data.y.float()
        else:
            preds = x.argmax(dim=-1)
            loss = self.loss_module(x, data.y)
            acc = (preds == data.y).sum().float() / preds.shape[0]
        return loss, acc

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(), lr=1e-2, weight_decay=0.0)  # High lr because of small dataset and small model
```

(continues on next page)
return optimizer

def training_step(self, batch, batch_idx):
    loss, acc = self.forward(batch, mode="train")
    self.log('train_loss', loss)
    self.log('train_acc', acc)
    return loss

def validation_step(self, batch, batch_idx):
    _, acc = self.forward(batch, mode="val")
    self.log('val_acc', acc)

def test_step(self, batch, batch_idx):
    _, acc = self.forward(batch, mode="test")
    self.log('test_acc', acc)

Below we train the model on our dataset. It resembles the typical training functions we have seen so far.

[26]: def train_graph_classifier(model_name, **model_kwargs):
    pl.seed_everything(42)
    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "GraphLevel" + model_name)
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                          checkpoint_callback=ModelCheckpoint(save_weights_only=True,
                          mode="max", monitor="val_acc"),
                          gpus=1 if str(device).startswith("cuda") else 0,
                          max_epochs=500,
                          progress_bar_refresh_rate=0)
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need
    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "GraphLevel%s.ckpt" % model_name)
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = GraphLevelGNN.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything(42)
        model = GraphLevelGNN(c_in=tu_dataset.num_node_features,
                               c_out=1 if tu_dataset.num_classes==2 else tu_dataset.
                               num_classes,
                               **model_kwargs)
        trainer.fit(model, graph_train_loader, graph_val_loader)
        model = GraphLevelGNN.load_from_checkpoint(trainer.checkpoint_callback.best_
        model_path)
    # Test best model on validation and test set
    train_result = trainer.test(model, test_dataloaders=graph_train_loader, verbose=False)
    test_result = trainer.test(model, test_dataloaders=graph_test_loader, verbose=False)
    result = {"test": test_result[0]["test_acc"], "train": train_result[0]["test_acc"],}
Finally, let's perform the training and testing. Feel free to experiment with different GNN layers, hyperparameters, etc.

```
model, result = train_graph_classifier(model_name="GraphConv",
    c_hidden=256,
    layer_name="GraphConv",
    num_layers=3,
    dp_rate_linear=0.5,
    dp_rate=0.0)
```

The test performance shows that we obtain quite good scores on an unseen part of the dataset. It should be noted that as we have been using the test set for validation as well, we might have overfitted slightly to this set. Nevertheless, the experiment shows us that GNNs can be indeed powerful to predict the properties of graphs and/or molecules.

### 5.9.4 Conclusion

In this tutorial, we have seen the application of neural networks to graph structures. We looked at how a graph can be represented (adjacency matrix or edge list), and discussed the implementation of common graph layers: GCN and GAT. The implementations showed the practical side of the layers, which is often easier than the theory. Finally, we experimented with different tasks, on node-, edge- and graph-level. Overall, we have seen that including graph information in the predictions can be crucial for achieving high performance. There are a lot of applications that benefit from GNNs, and the importance of these networks will likely increase over the next years.
5.10 Tutorial 8: Deep Energy-Based Generative Models

Filled notebook:

Pre-trained models:

In this tutorial, we will look at energy-based deep learning models, and focus on their application as generative models. Energy models have been a popular tool before the huge deep learning hype around 2012 hit. However, in recent years, energy-based models have gained increasing attention because of improved training methods and tricks being proposed. Although they are still in a research stage, they have shown to outperform strong Generative Adversarial Networks (Lecture/Tutorial 10) in certain cases which have been the state of the art of generating images (blog post about strong energy-based models, blog post about the power of GANs). Hence, it is important to be aware of energy-based models, and as the theory can be abstract sometimes, we will show the idea of energy-based models with a lot of examples.

First, let’s import our standard libraries below.

```python
# Standard libraries
import os
import json
import math
import numpy as np
import random

# Imports for plotting
import matplotlib.pyplot as plt
from matplotlib import cm
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
from mpl_toolkits.mplot3d.axes3d import Axes3D
from mpl_toolkits.mplot3d import proj3d
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

# PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
# Torchvision
import torchvision
from torchvision.datasets import MNIST
from torchvision import transforms
# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError:
    # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install pytorch-lightning==1.0.3
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
```

(continues on next page)
# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"

# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial8"

# Setting the seed
pl.seed_everything(42)

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

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

We also have pre-trained models that we download below.

```python
[2]:
import urllib.request
from urllib.error import HTTPError

# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial8/"

# Files to download
pretrained_files = ["MNIST.ckpt", "tensorboards/events.out.tfevents.MNIST"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/",1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print("Downloading $s...
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive folder, or contact the author with the full output including the following error:", e)
```

## 5.10.1 Energy Models

In the first part of this tutorial, we will review the theory of the energy-based models (the same theory has been discussed in Lecture 8). While most of the previous models had the goal of classification or regression, energy-based models are motivated from a different perspective: density estimation. Given a dataset with a lot of elements, we want to estimate the probability distribution over the whole data space. As an example, if we model images from CIFAR10, our goal would be to have a probability distribution over all possible images of size $32 \times 32 \times 3$ where those images have a high likelihood that look realistic and are one of the 10 CIFAR classes. Simple methods like interpolation between images don’t work because images are extremely high-dimensional (especially for large HD images). Hence, we turn to deep learning methods that have performed well on complex data.

However, how do we predict a probability distribution $p(x)$ over so many dimensions using a simple neural network? The problem is that we cannot just predict a score between 0 and 1, because a probability distribution over data needs
to fulfill two properties:

1. The probability distribution needs to assign any possible value of $x$ a non-negative value: $p(x) \geq 0$.
2. The probability density must sum/integrate to 1 over all possible inputs: $\int p(x)dx = 1$.

Luckily, there are actually many approaches for this, and one of them are energy-based models. The fundamental idea of energy-based models is that you can turn any function that predicts values larger than zero into a probability distribution by dividing by its volume. Imagine we have a neural network, which has as output a single neuron, like in regression. We can call this network $E_\theta(x)$, where $\theta$ are our parameters of the network, and $x$ the input data (e.g. an image). The output of $E_\theta$ is a scalar value between $-\infty$ and $\infty$. Now, we can use basic probability theory to normalize the scores of all possible inputs:

$$q_\theta(x) = \frac{\exp (-E_\theta(x))}{Z_\theta}$$

where $Z_\theta = \left\{ \begin{array}{ll} \int_x \exp (-E_\theta(x)) dx & \text{if } x \text{ is continuous} \\ \sum_x \exp (-E_\theta(x)) & \text{if } x \text{ is discrete} \end{array} \right.$

The exp-function ensures that we assign a probability greater than zero to any possible input. We use a negative sign in front of $E$ because we call $E_\theta$ to be the energy function: data points with high likelihood have a low energy, while data points with low likelihood have a high energy. $Z_\theta$ is our normalization terms that ensures that the density integrates/sums to 1. We can show this by integrating over $q_\theta(x)$:

$$\int_x q_\theta(x)dx = \int_x \frac{\exp (-E_\theta(x))}{Z_\theta} dx = \int_x \frac{\exp (-E_\theta(x))}{\sum_x \exp (-E_\theta(x))} dx = 1$$

Note that we call the probability distribution $q_\theta(x)$ because this is the learned distribution by the model, and is trained to be as close as possible to the true, unknown distribution $p(x)$.

The main benefit of this formulation of the probability distribution is its great flexibility as we can choose $E_\theta$ in whatever way we like, without any constraints. Nevertheless, when looking at the equation above, we can see a fundamental issue: How do we calculate $Z_\theta$? There is no chance that we can calculate $Z_\theta$ analytically for high-dimensional input and/or larger neural networks, but the task requires us to know $Z_\theta$. Although we can’t determine the exact likelihood of a point, there exist methods with which we can train energy-based models. Thus, we will look next at “Contrastive Divergence” for training the model.

**Contrastive Divergence**

When we train a model on generative modeling, it is usually done by maximum likelihood estimation. In other words, we try to maximize the likelihood of the examples in the training set. As the exact likelihood of a point cannot be determined due to the unknown normalization constant $Z_\theta$, we need to train energy-based models slightly different. We cannot just maximize the un-normalized probability $\exp(-E_\theta(x_{\text{train}}))$ because there is no guarantee that $Z_\theta$ stays constant, or that $x_{\text{train}}$ is becoming more likely than the others. However, if we base our training on comparing the likelihood of points, we can create a stable objective. Namely, we can re-write our maximum likelihood objective where we maximize the probability of $x_{\text{train}}$ compared to a randomly sampled data point of our model:

$$\nabla_\theta \mathcal{L}_{\text{MLE}}(\theta; p) = -\mathbb{E}_{p(x)} [\nabla_\theta \log q_\theta(x)] = \mathbb{E}_{p(x)} [\nabla_\theta E_\theta(x)] - \mathbb{E}_{q_\theta(x)} [\nabla_\theta E_\theta(x)]$$

Note that the loss is still an objective we want to minimize. Thus, we try to minimize the energy for data points from the dataset, while maximizing the energy for randomly sampled data points from our model (how we sample will be explained below). Although this objective sounds intuitive, how is it actually derived from our original distribution $q_\theta(x)$? The trick is that we approximate $Z_\theta$ by a single Monte-Carlo sample. This gives us the exact same objective as written above.

Visually, we can look at the objective as follows (figure credit - Stefano Ermon and Aditya Grover):
\( f_\theta \) represents \( \exp(-E_\theta(x)) \) in our case. The point on the right, called “correct answer”, represents a data point from the dataset (i.e. \( x_{\text{train}} \)), and the left point, “wrong answer”, a sample from our model (i.e. \( x_{\text{sample}} \)). Thus, we try to “pull up” the probability of the data points in the dataset, while “pushing down” randomly sampled points. The two forces for pulling and pushing are in balance iff \( q_\theta(x) = p(x) \).

**Sampling from Energy-Based Models**

For sampling from an energy-based model, we can apply a Markov Chain Monte Carlo using Langevin Dynamics. The idea of the algorithm is to start from a random point, and slowly move towards the direction of higher probability using the gradients of \( E_\theta \). Nevertheless, this is not enough to fully capture the probability distribution. We need to add noise \( \omega \) at each gradient step to the current sample. Under certain conditions such as that we perform the gradient steps an infinite amount of times, we would be able to create an exact sample from our modeled distribution. However, as this is not practically possible, we usually limit the chain to \( K \) steps (\( K \) a hyperparameter that needs to be finetuned). Overall, the sampling procedure can be summarized in the following algorithm:

**Applications of Energy-based models beyond generation**

Modeling the probability distribution for sampling new data is not the only application of energy-based models. Any application which requires us to compare two elements is much simpler to learn because we just need to go for the higher energy. A couple of examples are shown below (figure credit - Stefano Ermon and Aditya Grover). A classification setup like object recognition or sequence labeling can be considered as an energy-based task as we just need to find the \( Y \) input that minimizes the output \( E(X, Y) \) (hence maximizes probability). Similarly, a popular application of energy-based models is denoising of images. Given an image \( X \) with a lot of noise, we try to minimize the energy by finding the true input image \( Y \).

Nonetheless, we will focus on generative modeling here as in the next couple of lectures, we will discuss more generative deep learning approaches.

**5.10.2 Image generation**

As an example for energy-based models, we will train a model on image generation. Specifically, we will look at how we can generate MNIST digits with a very simple CNN model. However, it should be noted that energy models are not easy to train and often diverge if the hyperparameters are not well tuned. We will rely on training tricks proposed in the paper *Implicit Generation and Generalization in Energy-Based Models* by Yilun Du and Igor Mordatch (blog). The important part of this notebook is however to see how the theory above can actually be used in a model.

**Dataset**

First, we can load the MNIST dataset below. Note that we need to normalize the images between -1 and 1 instead of mean 0 and std 1 because during sampling, we have to limit the input space. Scaling between -1 and 1 makes it easier to implement it.

```
# Transformations applied on each image => make them a tensor and normalize between -1 and 1
transform = transforms.Compose([transforms.ToTensor(),
                               transforms.Normalize((0.5,), (0.5,))])
```

(continues on next page)
# Loading the training dataset. We need to split it into a training and validation part.

```python
train_set = MNIST(root=DATASET_PATH, train=True, transform=transform, download=True)
```

# Loading the test set

```python
test_set = MNIST(root=DATASET_PATH, train=False, transform=transform, download=True)
```

# We define a set of data loaders that we can use for various purposes later.
# Note that for actually training a model, we will use different data loaders with a lower batch size.

```python
train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True, num_workers=4, pin_memory=True)
```

```python
test_loader = data.DataLoader(test_set, batch_size=256, shuffle=False, drop_last=False, num_workers=4)
```

## CNN Model

First, we implement our CNN model. The MNIST images are of size 28x28, hence we only need a small model. As an example, we will apply several convolutions with stride 2 that downscale the images. If you are interested, you can also use a deeper model such as a small ResNet, but for simplicity, we will stick with the tiny network.

It is a good practice to use a smooth activation function like Swish instead of ReLU in the energy model. This is because we will rely on the gradients we get back with respect to the input image, which should not be sparse.

```python
[4]: class Swish(nn.Module):
    def forward(self, x):
        return x * torch.sigmoid(x)

class CNNModel(nn.Module):
    def __init__(self, hidden_features=32, out_dim=1, **kwargs):
        super().__init__()
        # We increase the hidden dimension over layers. Here pre-calculated for simplicity.
        c_hid1 = hidden_features//2
        c_hid2 = hidden_features
        c_hid3 = hidden_features*2

        # Series of convolutions and Swish activation functions
        self.cnn_layers = nn.Sequential(
            nn.Conv2d(1, c_hid1, kernel_size=5, stride=2, padding=4), # [16x16] - Larger padding to get 32x32 image
            Swish(),
            nn.Conv2d(c_hid1, c_hid2, kernel_size=3, stride=2, padding=1), # [8x8]
            Swish(),
            nn.Conv2d(c_hid2, c_hid3, kernel_size=3, stride=2, padding=1), # [4x4]
            Swish(),
            nn.Conv2d(c_hid3, c_hid3, kernel_size=3, stride=2, padding=1), # [2x2]
            Swish(),
            nn.Flatten(),
            nn.Linear(c_hid3*4, c_hid3),
            Swish(),
            nn.Linear(c_hid3, out_dim)
```

(continues on next page)
In the rest of the notebook, the output of the model will actually not represent $E_\theta(x)$, but $-E_\theta(x)$. This is a standard implementation practice for energy-based models, as some people also write the energy probability density as $q_\theta(x) = \exp(-f_\theta(x))$. In that case, the model would actually represent $f_\theta(x)$. In the training loss etc., we need to be careful to not switch up the signs.

### Sampling buffer

In the next part, we look at the training with sampled elements. To use the contrastive divergence objective, we need to generate samples during training. Previous work has shown that due to the high dimensionality of images, we need a lot of iterations inside the MCMC sampling to obtain reasonable samples. However, there is a training trick that significantly reduces the sampling cost: using a sampling buffer. The idea is that we store the samples of the last couple of batches in a buffer, and re-use those as the starting point of the MCMC algorithm for the next batches. This reduces the sampling cost because the model requires a significantly lower number of steps to converge to reasonable samples. However, to not solely rely on previous samples and allow novel samples as well, we re-initialize 5% of our samples from scratch (random noise between -1 and 1).

Below, we implement the sampling buffer. The function `sample_new_exmps` returns a new batch of “fake” images. We refer to those as fake images because they have been generated, but are not actually part of the dataset. As mentioned before, we use initialize 5% randomly, and 95% are randomly picked from our buffer. On this initial batch, we perform MCMC for 60 iterations to improve the image quality and come closer to samples from $q_\theta(x)$. In the function `generate_samples`, we implemented the MCMC for images. Note that the hyperparameters of `step_size`, `steps`, the noise standard deviation $\sigma$ are specifically set for MNIST, and need to be finetuned for a different dataset if you want to use such.

```python
[5]: class Sampler:
    def __init__(self, model, img_shape, sample_size, max_len=8192):
        """
        Inputs:
        model - Neural network to use for modeling E_theta
        img_shape - Shape of the images to model
        sample_size - Batch size of the samples
        max_len - Maximum number of data points to keep in the buffer
        """
        super().__init__()
        self.model = model
        self.img_shape = img_shape
        self.sample_size = sample_size
        self.max_len = max_len
        self.examples = [(torch.rand((1,)+img_shape)*2-1) for _ in range(self.sample_size)]

    def sample_new_exmps(self, steps=60, step_size=10):
        """
        Function for getting a new batch of "fake" images.
        Inputs:
        steps - Number of iterations in the MCMC algorithm
        """
        ...
step_size - Learning rate $\nu$ in the algorithm above

# Choose 95% of the batch from the buffer, 5% generate from scratch
n_new = np.random.binomial(self.sample_size, 0.05)
rand_imgs = torch.rand((n_new,) + self.img_shape) * 2 - 1
old_imgs = torch.cat(random.choices(self.examples, k=self.sample_size-n_new), dim=0)
inp_imgs = torch.cat([rand_imgs, old_imgs], dim=0).detach().to(device)

# Perform MCMC sampling
inp_imgs = Sampler.generate_samples(self.model, inp_imgs, steps=steps, step_size=step_size)

# Add new images to the buffer and remove old ones if needed
self.examples = list(inp_imgs.to(torch.device("cpu")).chunk(self.sample_size, dim=0)) + self.examples
self.examples = self.examples[:self.max_len]
return inp_imgs

@staticmethod
def generate_samples(model, inp_imgs, steps=60, step_size=10, return_img_per_step=False):
    ""
    Function for sampling images for a given model.
    Inputs:
    model - Neural network to use for modeling $E_{\theta}$
inp_imgs - Images to start from for sampling. If you want to generate new images, enter noise between -1 and 1.
steps - Number of iterations in the MCMC algorithm.
step_size - Learning rate $\nu$ in the algorithm above
return_img_per_step - If True, we return the sample at every iteration of the MCMC
    ""
    # Before MCMC: set model parameters to "required_grad=False"
    # because we are only interested in the gradients of the input.
is_training = model.training
model.eval()
for p in model.parameters():
    p.requires_grad = False
inp_imgs.requires_grad = True

    # We use a buffer tensor in which we generate noise each loop iteration.
    # More efficient than creating a new tensor every iteration.
    noise = torch.randn(inp_imgs.shape, device=inp_imgs.device)

    # List for storing generations at each step (for later analysis)
    imgs_per_step = []

    # Loop over K (steps)
    for _ in range(steps):
        # Part 1: Add noise to the input.
        noise.normal_(0, 0.005)
inp_imgs.data.add_(noise.data)
inp_imgs.data.clamp_(min=-1.0, max=1.0)

        # Part 2: calculate gradients for the current input.
        out_imgs = -model(inp_imgs)
out_imgs.sum().backward()
inp_imgs.grad.data.clamp_(-0.03, 0.03) # For stabilizing and preventing too high gradients

# Apply gradients to our current samples
inp_imgs.data.add_(-step_size * inp_imgs.grad.data)
inp_imgs.grad.detach_()
inp_imgs.grad.zero_()
inp_imgs.data.clamp_(min=-1.0, max=1.0)

if return_img_per_step:
    imgs_per_step.append(inp_imgs.clone().detach())

# Reactivate gradients for parameters for training
for p in model.parameters():
    p.requires_grad = True
model.train(is_training)

if return_img_per_step:
    return torch.stack(imgs_per_step, dim=0)
else:
    return inp_imgs

The idea of the buffer becomes a bit clearer in the following algorithm.

Training algorithm

With the sampling buffer being ready, we can complete our training algorithm. Below is shown a summary of the full training algorithm of an energy model on image modeling:

The first few statements in each training iteration concern the sampling of the real and fake data, as we have seen above with the sample buffer. Next, we calculate the contrastive divergence objective using our energy model $E_\theta$. However, one additional training trick we need is to add a regularization loss on the output of $E_\theta$. As the output of the network is not constrained and adding a large bias or not to the output doesn’t change the contrastive divergence loss, we need to ensure somehow else that the output values are in a reasonable range. Without the regularization loss, the output values will fluctuate in a very large range. With this, we ensure that the values for the real data are around 0, and the fake data likely slightly lower (for noise or outliers the score can be still significantly lower). As the regularization loss is less important than the Contrastive Divergence, we have a weight factor $\alpha$ which is usually quite some smaller than 1. Finally, we perform an update step with an optimizer on the combined loss and add the new samples to the buffer.

Below, we put this training dynamic into a PyTorch Lightning module:

```python
import torch

class DeepEnergyModel(pl.LightningModule):
    def __init__(self, img_shape, batch_size, alpha=0.1, lr=1e-4, betal=0.0, **CNN_args):
        super().__init__()
        self.save_hyperparameters()
        self.save_hyperparameters()

        self.cnn = CNNModel(**CNN_args)
        self.sampler = Sampler(self.cnn, img_shape=img_shape, sample_size=batch_size)
        self.example_input_array = torch.zeros(1, *img_shape)
```

(continues on next page)
def forward(self, x):
    z = self.cnn(x)
    return z

def configure_optimizers(self):
    # Energy models can have issues with momentum as the loss surfaces changes
    # with its parameters.
    # Hence, we set it to 0 by default.
    optimizer = optim.Adam(self.parameters(), lr=self.hparams.lr, betas=(self.
    -hparams.beta1, 0.999))
    scheduler = optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.97)  # Exponential
    # decay over epochs
    return [optimizer], [scheduler]

def training_step(self, batch, batch_idx):
    # We add minimal noise to the original images to prevent the model from
    # focusing on purely "clean" inputs
    real_imgs, _ = batch
    small_noise = torch.randn_like(real_imgs) * 0.005
    real_imgs.add_(small_noise).clamp_(min=-1.0, max=1.0)
    # Obtain samples
    fake_imgs = self.sampler.sample_new_exmps(steps=60, step_size=10)
    # Predict energy score for all images
    inp_imgs = torch.cat([real_imgs, fake_imgs], dim=0)
    real_out, fake_out = self.cnn(inp_imgs).chunk(2, dim=0)
    # Calculate losses
    reg_loss = self.hparams.alpha * (real_out ** 2 + fake_out ** 2).mean()
    cdiv_loss = fake_out.mean() - real_out.mean()
    loss = reg_loss + cdiv_loss
    # Logging
    self.log('loss', loss)
    self.log('loss_regularization', reg_loss)
    self.log('loss_contrastive_divergence', cdiv_loss)
    self.log('metrics_avg_real', real_out.mean())
    self.log('metrics_avg_fake', fake_out.mean())
    return loss

def validation_step(self, batch, batch_idx):
    # For validating, we calculate the contrastive divergence between purely
    # random images and unseen examples
    # Note that the validation/test step of energy-based models depends on what
    # we are interested in the model
    real_imgs, _ = batch
    fake_imgs = torch.rand_like(real_imgs) * 2 - 1
    inp_imgs = torch.cat([real_imgs, fake_imgs], dim=0)
    real_out, fake_out = self.cnn(inp_imgs).chunk(2, dim=0)
    cdiv = fake_out.mean() - real_out.mean()
    self.log('val_contrastive_divergence', cdiv)
We do not implement a test step because energy-based, generative models are usually not evaluated on a test set. The validation step however is used to get an idea of the difference between energy/likelihood of random images to unseen examples of the dataset.

**Callbacks**

To track the performance of our model during training, we will make extensive use of PyTorch Lightning’s callback framework. Remember that callbacks can be used for running small functions at any point of the training, for instance after finishing an epoch. Here, we will use three different callbacks we define ourselves.

The first callback, called `GenerateCallback`, is used for adding image generations to the model during training. After every $N$ epochs (usually $N = 5$ to reduce output to TensorBoard), we take a small batch of random images and perform many MCMC iterations until the model’s generation converges. Compared to the training that used 60 iterations, we use 256 here because (1) we only have to do it once compared to the training that has to do it every iteration, and (2) we do not start from a buffer here, but from scratch. It is implemented as follows:

```python
class GenerateCallback(pl.Callback):
    def __init__(self, batch_size=8, vis_steps=8, num_steps=256, every_n_epochs=5):
        super().__init__()  # Number of images to generate
        self.batch_size = batch_size
        self.vis_steps = vis_steps  # Number of steps within generation to visualize
        self.num_steps = num_steps  # Number of steps to take during generation
        self.every_n_epochs = every_n_epochs  # Only save those images every N epochs
        # otherwise tensorboard gets quite large

    def on_epoch_end(self, trainer, pl_module):
        if trainer.current_epoch % self.every_n_epochs == 0:
            # Generate images
            imgs_per_step = self.generate_imgs(pl_module)
            # Plot and add to tensorboard
            for i in range(imgs_per_step.shape[1]):
                step_size = self.num_steps // self.vis_steps
                imgs_to_plot = imgs_per_step[step_size-1::step_size,i]
                grid = torchvision.utils.make_grid(imgs_to_plot, nrow=imgs_to_plot.shape[0], normalize=True, range=(-1,1))
                trainer.logger.experiment.add_image("generation_%i" % i, grid, global_step=trainer.current_epoch)

    def generate_imgs(self, pl_module):
        pl_module.eval()
        start_imgs = torch.randn((self.batch_size,) + pl_module.hparams["img_shape"]).to(pl_module.device)
        imgs_per_step = Sampler.generate_samples(pl_module.cnn, start_imgs,
                                                steps=self.num_steps, step_size=10, return_img_per_step=True)
        pl_module.train()
        return imgs_per_step
```
The second callback is called SamplerCallback, and simply adds a randomly picked subset of images in the sampling buffer to the TensorBoard. This helps to understand what images are currently shown to the model as “fake”.

```python
[8]: class SamplerCallback(pl.Callback):
    def __init__(self, num_imgs=32, every_n_epochs=5):
        super().__init__()
        self.num_imgs = num_imgs  # Number of images to plot
        self.every_n_epochs = every_n_epochs  # Only save those images every N epochs
        # (otherwise tensorboard gets quite large)

    def on_epoch_end(self, trainer, pl_module):
        if trainer.current_epoch % self.every_n_epochs == 0:
            exmp_imgs = torch.cat(random.choices(pl_module.sampler.examples, k=self.num_imgs), dim=0)
            grid = torchvision.utils.make_grid(exmp_imgs, nrow=4, normalize=True, range=(-1,1))
            trainer.logger.experiment.add_image("sampler", grid, global_step=trainer.current_epoch)
```

Finally, our last callback is OutlierCallback. This callback evaluates the model by recording the (negative) energy assigned to random noise. While our training loss is almost constant across iterations, this score is likely showing the progress of the model to detect “outliers”.

```python
[9]: class OutlierCallback(pl.Callback):
    def __init__(self, batch_size=1024):
        super().__init__()
        self.batch_size = batch_size

    def on_epoch_end(self, self, trainer, pl_module):
        with torch.no_grad():
            pl_module.eval()
            rand_imgs = torch.rand((self.batch_size,) + pl_module.hparams["img_shape"] + (1,)).to(pl_module.device)
            rand_imgs = rand_imgs * 2 - 1.0
            rand_out = pl_module.cnn(rand_imgs).mean()
            pl_module.train()

            trainer.logger.experiment.add_scalar("rand_out", rand_out, global_step=trainer.current_epoch)
```

### Running the model

Finally, we can add everything together to create our final training function. The function is very similar to any other PyTorch Lightning training function we have seen so far. However, there is the small difference of that we do not test the model on a test set because we will analyse the model afterward by checking its prediction and ability to perform outlier detection.

```python
[10]: def train_model(**kwargs):
    # Create a PyTorch Lightning trainer with the generation callback
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "MNIST"),
                         checkpoint_callback=ModelCheckpoint(save_weights_only=True, mode="min", monitor='val_contrastive_divergence'),
                         gpus=1,
                         max_epochs=60,
```

(continues on next page)
gradient_clip_val=0.1,
callbacks=[GenerateCallback(every_n_epochs=5),
         SamplerCallback(every_n_epochs=5),
         OutlierCallback(),
         LearningRateMonitor("epoch")
         ],
progress_bar_refresh_rate=1)

# Check whether pretrained model exists. If yes, load it and skip training
pretrained_filename = os.path.join(CHECKPOINT_PATH, "MNIST.ckpt")
if os.path.isfile(pretrained_filename):
    print("Found pretrained model, loading...")
    model = DeepEnergyModel.load_from_checkpoint(pretrained_filename)
else:
    pl.seed_everything(42)
    model = DeepEnergyModel(**kwargs)
    trainer.fit(model, train_loader, test_loader)
    model = DeepEnergyModel.load_from_checkpoint(trainer.checkpoint_callback.best_
    model_path)

# No testing as we are more interested in other properties
return model

[11]: model = train_model(img_shape=(1,28,28),
                       batch_size=train_loader.batch_size,
                       lr=1e-4,
                       betal=0.0)

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Found pretrained model, loading...

5.10.3 Analysis

In the last part of the notebook, we will try to take the trained energy-based generative model, and analyse its properties.

TensorBoard

The first thing we can look at is the TensorBoard generate during training. This can help us to understand the training dynamic even better, and shows potential issues. Let’s load the TensorBoard below:

[12]: # Import tensorboard
    from torch.utils.tensorboard import SummaryWriter
    %load_ext tensorboard

[13]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH!
    %tensorboard --logdir ../saved_models/tutorial8/tensorboards/
We see that the contrastive divergence as well as the regularization converge quickly to 0. However, the training continues although the loss is always close to zero. This is because our “training” data changes with the model by sampling. The progress of training can be best measured by looking at the samples across iterations, and the score for random images that decreases constantly over time.

**Image Generation**

Another way of evaluating generative models is by sampling a few generated images. Generative models need to be good at generating realistic images as this truly shows that they have modeled the true data distribution. Thus, let’s sample a few images of the model below:

```python
[14]: model.to(device)
pl.seed_everything(43)
callback = GenerateCallback(batch_size=4, vis_steps=8, num_steps=256)
imgs_per_step = callback.generate_imgs(model)
imgs_per_step = imgs_per_step.cpu()
```

The characteristic of sampling with energy-based models is that they require the iterative MCMC algorithm. To gain an insight in how the images change over iterations, we plot a few intermediate samples in the MCMC as well:

```python
[15]:
    for i in range(imgs_per_step.shape[1]):
        step_size = callback.num_steps // callback.vis_steps
        imgs_to_plot = imgs_per_step[step_size-i::step_size,i]
        imgs_to_plot = torch.cat([imgs_per_step[0:1,i],imgs_to_plot], dim=0)
        grid = torchvision.utils.make_grid(imgs_to_plot, nrow=imgs_to_plot.shape[0],
                                            normalize=True, range=(-1,1), pad_value=0.5, padding=2)
        grid = grid.permute(1, 2, 0)
```

(continues on next page)
We see that although starting from noise in the very first step, the sampling algorithm obtains reasonable shapes after only 32 steps. Over the next 200 steps, the shapes become clearer and changed towards realistic digits. The specific samples can differ when you run the code on Colab, hence the following description is specific to the plots shown on the website. The first row shows an 8, where we remove unnecessary white parts over iterations. The transformation across iterations can be seen at best for the second sample, which creates a digit of 2. While the first sample after 32 iterations looks a bit like a digit, but not really, the sample is transformed more and more to a typical image of the digit 2.
Out-of-distribution detection

A very common and strong application of energy-based models is out-of-distribution detection (sometimes referred to as “anomaly” detection). As more and more deep learning models are applied in production and applications, a crucial aspect of these models is to know what the models don’t know. Deep learning models are usually overconfident, meaning that they classify even random images sometimes with 100% probability. Clearly, this is not something that we want to see in applications. Energy-based models can help with this problem because they are trained to detect images that do not fit the training dataset distribution. Thus, in those applications, you could train an energy-based model along with the classifier, and only output predictions if the energy-based models assign a (unnormalized) probability higher than $\delta$ to the image. You can actually combine classifiers and energy-based objectives in a single model, as proposed in this paper.

In this part of the analysis, we want to test the out-of-distribution capability of our energy-based model. Remember that a lower output of the model denotes a low probability. Thus, we hope to see low scores if we enter random noise to the model:

```python
with torch.no_grad():
    rand_imgs = torch.rand((128,) + model.hparams.img_shape).to(model.device)
    rand_imgs = rand_imgs * 2 - 1.0
    rand_out = model.cnn(rand_imgs).mean()
    print("Average score for random images: \$%.2f\" % (rand_out.item()))
```

Average score for random images: -17.88

As we hoped, the model assigns very low probability to those noisy images. As another reference, let’s look at predictions for a batch of images from the training set:

```python
with torch.no_grad():
    train_imgs, _ = next(iter(train_loader))
    train_imgs = train_imgs.to(model.device)
    train_out = model.cnn(train_imgs).mean()
    print("Average score for training images: \$%.2f\" % (train_out.item()))
```

Average score for training images: -0.00

The scores are close to 0 because of the regularization objective that was added to the training. So clearly, the model can distinguish between noise and real digits. However, what happens if we change the training images a little, and see which ones gets a very low score?

```python
@torch.no_grad()

def compare_images(img1, img2):
    imgs = torch.stack([img1, img2], dim=0).to(model.device)
    score1, score2 = model.cnn(imgs).cpu().chunk(2, dim=0)
    grid = torchvision.utils.make_grid([img1.cpu(), img2.cpu()], nrow=2, normalize=True, range=(-1,1), pad_value=0.5, padding=2)
    grid = grid.permute(1, 2, 0)
    plt.figure(figsize=(4,4))
    plt.imshow(grid)
    plt.xticks([(img1.shape[2]+2)*(0.5+j) for j in range(2)], labels=["Original image", "Transformed image"])  
    plt.yticks([])
    plt.show()
    print("Score original image: \$%.2f\" % score1)
    print("Score transformed image: \$%.2f\" % score2)
```

We use a random test image for this. Feel free to change it to experiment with the model yourself.
The first transformation is to add some random noise to the image:

```python
[20]:
    img_noisy = exmp_img + torch.randn_like(exmp_img) * 0.3
    img_noisy.clamp_(min=-1.0, max=1.0)
    compare_images(exmp_img, img_noisy)
```

We can see that the score considerably drops. Hence, the model can detect random Gaussian noise on the image. This is also to expect as initially, the “fake” samples are pure noise images.

Next, we flip an image and check how this influences the score:

```python
[21]:
    img_flipped = exmp_img.flip(dims=(1,2))
    compare_images(exmp_img, img_flipped)
```

If the digit can only be read in this way, for example, the 7, then we can see that the score drops. However, the score only drops slightly. This is likely because of the small size of our model. Keep in mind that generative modeling is a much harder task than classification, as we do not only need to distinguish between classes but learn all details/characteristics of the digits. With a deeper model, this could eventually be captured better (but at the cost of greater training instability).

Finally, we check what happens if we reduce the digit significantly in size:
The score again drops but not by a large margin, although digits in the MNIST dataset usually are much larger.

Overall, we can conclude that our model is good for detecting Gaussian noise and smaller transformations to existing digits. Nonetheless, to obtain a very good out-of-distribution model, we would need to train deeper models and for more iterations.

**Instability**

Finally, we should discuss the possible instabilities of energy-based models, in particular for the example of image generation that we have implemented in this notebook. In the process of hyperparameter search for this notebook, there have been several models that diverged. Divergence in energy-based models means that the models assign a high probability to examples of the training set which is a good thing. However, at the same time, the sampling algorithm fails and only generates noise images that obtain minimal probability scores. This happens because the model has created many local maxima in which the generated noise images fall. The energy surface over which we calculate the gradients to reach data points with high probability has “diverged” and is not useful for our MCMC sampling.

Besides finding the optimal hyperparameters, a common trick in energy-based models is to reload stable checkpoints. If we detect that the model is diverging, we stop the training, load the model from one epoch ago where it did not diverge yet. Afterward, we continue training and hope that with a different seed the model is not diverging again. Nevertheless, this should be considered as the “last hope” for stabilizing the models, and careful hyperparameter tuning is the better way to do so. Sensitive hyperparameters include `step_size`, `steps` and the noise standard deviation in the sampler, and the learning rate and feature dimensionality in the CNN model.

**5.10.4 Conclusion**

In this tutorial, we have discussed energy-based models for generative modeling. The concept relies on the idea that any strictly positive function can be turned into a probability distribution by normalizing over the whole dataset. As this is not reasonable to calculate for high dimensional data like images, we train the model using contrastive divergence and sampling via MCMC. While the idea allows us to turn any neural network into an energy-based model, we have seen that there are multiple training tricks needed to stabilize the training. Furthermore, the training time of these models is relatively long as, during every training iteration, we need to sample new “fake” images, even with a sampling buffer. In the next lectures and assignment, we will see different generative models (e.g. VAE, GAN, NF) that allow us to do generative modeling more stably, but with the cost of more parameters.
5.11 Tutorial 9: Deep Autoencoders

Filled notebook:

Pre-trained models:

In this tutorial, we will take a closer look at autoencoders (AE). Autoencoders are trained on encoding input data such as images into a smaller feature vector, and afterward, reconstruct it by a second neural network, called a decoder. The feature vector is called the “bottleneck” of the network as we aim to compress the input data into a smaller amount of features. This property is useful in many applications, in particular in compressing data or comparing images on a metric beyond pixel-level comparisons. Besides learning about the autoencoder framework, we will also see the “deconvolution” (or transposed convolution) operator in action for scaling up feature maps in height and width. Such deconvolution networks are necessary wherever we start from a small feature vector and need to output an image of full size (e.g. in VAE, GANs, or super-resolution applications).

First of all, we again import most of our standard libraries. We will use PyTorch Lightning to reduce the training code overhead.

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

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

    # Progress bar
    from tqdm.notebook import tqdm

    # PyTorch
    import torch
    import torch.nn as nn
    import torch.nn.functional as F
    import torch.utils.data as data
    import torch.optim as optim
    # Torchvision
    import torchvision
    from torchvision.datasets import CIFAR10
    from torchvision import transforms
    # PyTorch Lightning
    try:
        import pytorch_lightning as pl
    except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
        !pip install pytorch-lightning==1.0.3
        import pytorch_lightning as pl
```

(continues on next page)
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Tensorboard extension (for visualization purposes later)
from torch.utils.tensorboard import SummaryWriter

%load_ext tensorboard

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"

# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial9"

# Setting the seed
pl.seed_everything(42)

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

We have 4 pretrained models that we have to download. Remember to adjust the variables DATASET_PATH and CHECKPOINT_PATH if needed.

[2]:
import urllib.request
from urllib.error import HTTPError

# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial9/"

# Files to download
pretrained_files = ["cifar10_64.ckpt", "cifar10_128.ckpt", "cifar10_256.ckpt", "cifar10_384.ckpt"]

# Create checkpoint path if it doesn’t exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

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

In this tutorial, we work with the CIFAR10 dataset. In CIFAR10, each image has 3 color channels and is 32x32 pixels large. As autoencoders do not have the constrain of modeling images probabilistic, we can work on more complex image data (i.e. 3 color channels instead of black-and-white) much easier than for VAEs. In case you have downloaded CIFAR10 already in a different directory, make sure to set DATASET_PATH accordingly to prevent another download.

In contrast to previous tutorials on CIFAR10 like Tutorial 5 (CNN classification), we do not normalize the data explicitly with a mean of 0 and std of 1, but roughly estimate it scaling the data between -1 and 1. This is because limiting the range will make our task of predicting/reconstructing images easier.

[3]:
# Transformations applied on each image => only make them a tensor
transform = transforms.Compose([transforms.ToTensor(),
                             transforms.Normalize((0.5,), (0.5,))])

5.11. Tutorial 9: Deep Autoencoders 193
# Loading the training dataset. We need to split it into a training and validation part
train_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=transform, 
  download=True)
pl.seed_everything(42)
train_set, val_set = torch.utils.data.random_split(train_dataset, [45000, 5000])

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

# We define a set of data loaders that we can use for various purposes later.
train_loader = data.DataLoader(train_set, batch_size=256, shuffle=True, drop_last=True, pin_memory=True, num_workers=4)
val_loader = data.DataLoader(val_set, batch_size=256, shuffle=False, drop_last=False, num_workers=4)
test_loader = data.DataLoader(test_set, batch_size=256, shuffle=False, drop_last=False, num_workers=4)

def get_train_images(num):
  return torch.stack([train_dataset[i][0] for i in range(num)], dim=0)

5.11.1 Building the autoencoder

In general, an autoencoder consists of an encoder that maps the input \( x \) to a lower-dimensional feature vector \( z \), and a decoder that reconstructs the input \( \hat{x} \) from \( z \). We train the model by comparing \( x \) to \( \hat{x} \) and optimizing the parameters to increase the similarity between \( x \) and \( \hat{x} \). See below for a small illustration of the autoencoder framework.

We first start by implementing the encoder. The encoder effectively consists of a deep convolutional network, where we scale down the image layer-by-layer using strided convolutions. After downscaling the image three times, we flatten the features and apply linear layers. The latent representation \( z \) is therefore a vector of size \( d \) which can be flexibly selected.

```python
class Encoder(nn.Module):
  def __init__(self, num_input_channels : int, base_channel_size : int, latent_dim : int, act_fn : object = nn.GELU):
    """Inputs:
      - num_input_channels : Number of input channels of the image. For CIFAR, this parameter is 3
      - base_channel_size : Number of channels we use in the first convolutional layers. Deeper layers might use a duplicate of it.
      - latent_dim : Dimensionality of latent representation z
      - act_fn : Activation function used throughout the encoder network
    """
    super().__init__()
    c_hid = base_channel_size
    # (continues on next page)
```

(continued from previous page)
self.net = nn.Sequential(
    nn.Conv2d(num_input_channels, c_hid, kernel_size=3, padding=1, stride=2),
    # 32x32 => 16x16
    act_fn(),
    nn.Conv2d(c_hid, c_hid, kernel_size=3, padding=1),
    act_fn(),
    nn.Conv2d(c_hid, 2*c_hid, kernel_size=3, padding=1, stride=2),
    # 16x16 => 8x8
    act_fn(),
    nn.Conv2d(2*c_hid, 2*c_hid, kernel_size=3, padding=1),
    act_fn(),
    nn.Conv2d(2*c_hid, 2*c_hid, kernel_size=3, padding=1, stride=2),
    # 8x8 => 4x4
    act_fn(),
    nn.Flatten(),
    # Image grid to single feature vector
    nn.Linear(2*16*c_hid, latent_dim)
)

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

Note that we do not apply Batch Normalization here. This is because we want the encoding of each image to be independent of all the other images. Otherwise, we might introduce correlations into the encoding or decoding that we do not want to have. In some implementations, you still can see Batch Normalization being used, because it can also serve as a form of regularization. Nevertheless, the better practice is to go with other normalization techniques if necessary like Instance Normalization or Layer Normalization. Given the small size of the model, we can neglect normalization for now.

The decoder is a mirrored, flipped version of the encoder. The only difference is that we replace strided convolutions by transposed convolutions (i.e. deconvolutions) to upscale the features. Transposed convolutions can be imagined as adding the stride to the input instead of the output, and can thus upscale the input. For an illustration of a `nn.ConvTranspose2d` layer with kernel size 3, stride 2, and padding 1, see below (figure credit - Vincent Dumoulin and Francesco Visin):

You see that for an input of size $3 \times 3$, we obtain an output of $5 \times 5$. However, to truly have a reverse operation of the convolution, we need to ensure that the layer scales the input shape by a factor of 2 (e.g. $4 \times 4 \rightarrow 8 \times 8$). For this, we can specify the parameter `output_padding` which adds additional values to the output shape. Note that we do not perform zero-padding with this, but rather increase the output shape for calculation.

Overall, the decoder can be implemented as follows:

```
[5]: class Decoder(nn.Module):
    
def __init__(self,
        num_input_channels : int,
        base_channel_size : int,
        latent_dim : int,
        act_fn : object = nn.GELU):
        
        # For CIFAR, this parameter is 3
        # layers. Early layers might use a duplicate of it.
```
The encoder and decoder networks we chose here are relatively simple. Usually, more complex networks are applied, especially when using a ResNet-based architecture. For example, see VQ-VAE and NVAE (although the papers discuss architectures for VAEs, they can equally be applied to standard autoencoders).

In a final step, we add the encoder and decoder together into the autoencoder architecture. We define the autoencoder as PyTorch Lightning Module to simplify the needed training code:

```python
class Autoencoder(pl.LightningModule):
    def __init__(self,
                 base_channel_size: int,
                 latent_dim: int,
                 encoder_class : object = Encoder,
                 decoder_class : object = Decoder,
                 num_input_channels: int = 3,
                 width: int = 32,
                 height: int = 32):
        super().__init__()
        # Saving hyperparameters of autoencoder
        self.save_hyperparameters()
        # Creating encoder and decoder
        self.encoder = encoder_class(num_input_channels, base_channel_size, latent_dim)
        self.decoder = decoder_class(num_input_channels, base_channel_size, latent_dim)
        # Example input array needed for visualizing the graph of the network
```

(continues on next page)
self.example_input_array = torch.zeros(2, num_input_channels, width, height)

def forward(self, x):
    
    The forward function takes in an image and returns the reconstructed image
    
    z = self.encoder(x)
    x_hat = self.decoder(z)
    return x_hat

def _get_reconstruction_loss(self, batch):
    
    Given a batch of images, this function returns the reconstruction loss (MSE, in our case)
    
    x, _ = batch  # We do not need the labels
    x_hat = self.forward(x)
    loss = F.mse_loss(x, x_hat, reduction="none")
    loss = loss.sum(dim=[1,2,3]).mean(dim=[0])
    return loss

def configure_optimizers(self):
    optimizer = optim.Adam(self.parameters(), lr=1e-3)
    # Using a scheduler is optional but can be helpful.
    # The scheduler reduces the LR if the validation performance hasn't improved for the last N epochs
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
        mode='min',
        factor=0.2,
        patience=20,
        min_lr=5e-5)
    return {
        "optimizer": optimizer,
        "lr_scheduler": scheduler,
        "monitor": "val_loss"
    }

def training_step(self, batch, batch_idx):
    loss = self._get_reconstruction_loss(batch)
    self.log('train_loss', loss)
    return loss

def validation_step(self, batch, batch_idx):
    loss = self._get_reconstruction_loss(batch)
    self.log('val_loss', loss)

def test_step(self, batch, batch_idx):
    loss = self._get_reconstruction_loss(batch)
    self.log('test_loss', loss)

For the loss function, we use the mean squared error (MSE). The mean squared error pushes the network to pay special attention to those pixel values its estimate is far away. Predicting 127 instead of 128 is not important when reconstructing, but confusing 0 with 128 is much worse. Note that in contrast to VAEs, we do not predict the probability per pixel value, but instead use a distance measure. This saves a lot of parameters and simplifies training. To get a better intuition per pixel, we report the summed squared error averaged over the batch dimension (any other mean/sum leads to the same result/parameters).

However, MSE has also some considerable disadvantages. Usually, MSE leads to blurry images where small noise/high-frequent patterns are removed as those cause a very low error. To ensure realistic images to be reconstructed, one could combine Generative Adversarial Networks (lecture 10) with autoencoders as done in several works.
(e.g. see here, here or these slides). Additionally, comparing two images using MSE does not necessarily reflect their visual similarity. For instance, suppose the autoencoder reconstructs an image shifted by one pixel to the right and bottom. Although the images are almost identical, we can get a higher loss than predicting a constant pixel value for half of the image (see code below). An example solution for this issue includes using a separate, pre-trained CNN, and use a distance of visual features in lower layers as a distance measure instead of the original pixel-level comparison.

```python
[7]: def compare_imgs(img1, img2, title_prefix=""):  
    # Calculate MSE loss between both images  
    loss = F.mse_loss(img1, img2, reduction="sum")  
    # Plot images for visual comparison
    grid = torchvision.utils.make_grid(torch.stack([img1, img2], dim=0), nrow=2,  
        normalize=True, range=(-1,1))
    grid = grid.permute(1, 2, 0)  
    plt.figure(figsize=(4,2))  
    plt.title("%s Loss: %4.2f" % (title_prefix, loss.item()))
    plt.imshow(grid)
    plt.axis('off')
    plt.show()

    for i in range(2):
        # Load example image
        img, _ = train_dataset[i]
        img_mean = img.mean(dim=[1,2], keepdims=True)

        # Shift image by one pixel
        SHIFT = 1
        img_shifted = torch.roll(img, shifts=SHIFT, dims=1)
        img_shifted = torch.roll(img_shifted, shifts=SHIFT, dims=2)
        img_shifted[:,:1,:] = img_mean
        img_shifted[:,:,:1] = img_mean
        compare_imgs(img, img_shifted, "Shifted -")

        # Set half of the image to zero
        img_masked = img.clone()
        img_masked[:,:img_masked.shape[1]//2,:] = img_mean
        compare_imgs(img, img_masked, "Masked -")
```

Shifted - Loss: 205.40
Training the model

During the training, we want to keep track of the learning progress by seeing reconstructions made by our model. For this, we implement a callback object in PyTorch Lightning which will add reconstructions every $N$ epochs to our tensorboard:

```python
[8]: class GenerateCallback(pl.Callback):
    
def __init__(self, input_imgs, every_n_epochs=1):
        super().__init__()
        self.input_imgs = input_imgs  # Images to reconstruct during training
        self.every_n_epochs = every_n_epochs  # Only save those images every N epochs
        
def on_epoch_end(self, trainer, pl_module):
        if trainer.current_epoch % self.every_n_epochs == 0:
```

(continues on next page)
# Reconstruct images
input_imgs = self.input_imgs.to(pl_module.device)
with torch.no_grad():
    pl_module.eval()
    reconst_imgs = pl_module(input_imgs)
pl_module.train()

# Plot and add to tensorboard
imgs = torch.stack([input_imgs, reconst_imgs], dim=1).flatten(0,1)
grid = torchvision.utils.make_grid(imgs, nrow=2, normalize=True, range=(-1,1))
trainer.logger.experiment.add_image("Reconstructions", grid, global_step=trainer.global_step)

We will now write a training function that allows us to train the autoencoder with different latent dimensionality and returns both the test and validation score. We provide pre-trained models and recommend you using those, especially when you work on a computer without GPU. Of course, feel free to train your own models on Lisa.

```python
[9]: def train_cifar(latent_dim):
    # Create a PyTorch Lightning trainer with the generation callback
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "cifar10_%i" % latent_dim),
                          checkpoint_callback=ModelCheckpoint(save_weights_only=True),
                          gpus=1,
                          max_epochs=500,
                          callbacks=[GenerateCallback(get_train_images(8), every_n_epochs=10), LearningRateMonitor("epoch")])
    trainer.logger._log_graph = True # If True, we plot the computation graph in tensorboard
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "cifar10_%i.ckpt" % latent_dim)
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = Autoencoder.load_from_checkpoint(pretrained_filename)
    else:
        model = Autoencoder(base_channel_size=32, latent_dim=latent_dim)
    trainer.fit(model, train_loader, val_loader)
    # Test best model on validation and test set
    val_result = trainer.test(model, test_dataloaders=val_loader, verbose=False)
    test_result = trainer.test(model, test_dataloaders=test_loader, verbose=False)
    result = {"test": test_result, "val": val_result}
    return model, result
```
Comparing latent dimensionality

When training an autoencoder, we need to choose a dimensionality for the latent representation $z$. The higher the latent dimensionality, the better we expect the reconstruction to be. However, the idea of autoencoders is to compress data. Hence, we are also interested in keeping the dimensionality low. To find the best tradeoff, we can train multiple models with different latent dimensionalities. The original input has $32 \times 32 \times 3 = 3072$ pixels. Keeping this in mind, a reasonable choice for the latent dimensionality might be between 64 and 384:

```python
[10]: model_dict = {}
    for latent_dim in [64, 128, 256, 384]:
        model_ld, result_ld = train_cifar(latent_dim)
        model_dict[latent_dim] = {"model": model_ld, "result": result_ld}
```

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Found pretrained model, loading...
After training the models, we can plot the reconstruction loss over the latent dimensionality to get an intuition how these two properties are correlated:

```python
[11]: latent_dims = sorted([k for k in model_dict])
val_scores = [model_dict[k]["result"]['val'][0]['test_loss'] for k in latent_dims]

fig = plt.figure(figsize=(6,4))
plt.plot(latent_dims, val_scores, '--', color="#000", marker="*", markeredgecolor="#000", markerfacecolor="y", markersize=16)
plt.xscale("log")
plt.xticks(latent_dims, labels=latent_dims)
plt.title("Reconstruction error over latent dimensionality", fontsize=14)
plt.xlabel("Latent dimensionality")
plt.ylabel("Reconstruction error")
plt.minorticks_off()
plt.ylim(0,100)
plt.show()
```
As we initially expected, the reconstruction loss goes down with increasing latent dimensionality. For our model and setup, the two properties seem to be exponentially (or double exponentially) correlated. To understand what these differences in reconstruction error mean, we can visualize example reconstructions of the four models:

```python
[12]: def visualize_reconstructions(model, input_imgs):
    # Reconstruct images
    model.eval()
    with torch.no_grad():
        reconst_imgs = model(input_imgs.to(model.device))
        reconst_imgs = reconst_imgs.cpu()

    # Plotting
    imgs = torch.stack([input_imgs, reconst_imgs], dim=1).flatten(0,1)
    grid = torchvision.utils.make_grid(imgs, nrow=4, normalize=True, range=(-1,1))
    grid = grid.permute(1, 2, 0)
    plt.figure(figsize=(7,4.5))
    plt.title("Reconstructed from %i latents" % (model.hparams.latent_dim))
    plt.imshow(grid)
    plt.axis('off')
    plt.show()

[13]: input_imgs = get_train_images(4)
for latent_dim in model_dict:
    visualize_reconstructions(model_dict[latent_dim]["model"], input_imgs)
Reconstructed from 64 latents

Reconstructed from 128 latents
Clearly, the smallest latent dimensionality can only save information about the rough shape and color of the object, but the reconstructed image is extremely blurry and it is hard to recognize the original object in the reconstruction. With 128 features, we can recognize some shapes again although the picture remains blurry. The models with the highest two dimensionalities reconstruct the images quite well. The difference between 256 and 384 is marginal at first sight but can be noticed when comparing, for instance, the backgrounds of the first image (the 384 features model more of the pattern than 256).
Out-of-distribution images

Before continuing with the applications of autoencoder, we can actually explore some limitations of our autoencoder. For example, what happens if we try to reconstruct an image that is clearly out of the distribution of our dataset? We expect the decoder to have learned some common patterns in the dataset, and thus might in particular fail to reconstruct images that do not follow these patterns.

The first experiment we can try is to reconstruct noise. We, therefore, create two images whose pixels are randomly sampled from a uniform distribution over pixel values, and visualize the reconstruction of the model (feel free to test different latent dimensionalities):

```python
rand_imgs = torch.rand(2, 3, 32, 32) * 2 - 1
visualize_reconstructions(model_dict[256]["model"], rand_imgs)
```

The reconstruction of the noise is quite poor, and seems to introduce some rough patterns. As the input does not follow the patterns of the CIFAR dataset, the model has issues reconstructing it accurately.

We can also check how well the model can reconstruct other manually-coded patterns:

```python
plain_imgs = torch.zeros(4, 3, 32, 32)
# Single color channel
plain_imgs[1,0] = 1
# Checkboard pattern
plain_imgs[2,:,16:,16:] = 1
plain_imgs[2,:,16:,16:] = -1
# Color progression
xx, yy = torch.meshgrid(torch.linspace(-1,1,32), torch.linspace(-1,1,32))
plain_imgs[3,0,:,::] = xx
plain_imgs[3,1,:,::] = yy
visualize_reconstructions(model_dict[256]["model"], plain_imgs)
```
The plain, constant images are reconstructed relatively good although the single color channel contains some noticeable noise. The hard borders of the checkboard pattern are not as sharp as intended, as well as the color progression, both because such patterns never occur in the real-world pictures of CIFAR.

In general, autoencoders tend to fail reconstructing high-frequent noise (i.e. sudden, big changes across few pixels) due to the choice of MSE as loss function (see our previous discussion about loss functions in autoencoders). Small misalignments in the decoder can lead to huge losses so that the model settles for the expected value/mean in these regions. For low-frequent noise, a misalignment of a few pixels does not result in a big difference to the original image. However, the larger the latent dimensionality becomes, the more of this high-frequent noise can be accurately reconstructed.

**Generating new images**

Variational autoencoders are a generative version of the autoencoders because we regularize the latent space to follow a Gaussian distribution. However, in vanilla autoencoders, we do not have any restrictions on the latent vector. So what happens if we would actually input a randomly sampled latent vector into the decoder? Let’s find it out below:

```
[16]: model = model_dict[256]["model"]
latent_vectors = torch.randn(8, model.hparams.latent_dim, device=model.device)
with torch.no_grad():
    imgs = model.decoder(latent_vectors)
    imgs = imgs.cpu()
grid = torchvision.utils.make_grid(imgs, nrow=4, normalize=True, range=(-1,1), pad_value=0.5)
grid = grid.permute(1, 2, 0)
plt.figure(figsize=(8,5))
plt.imshow(grid)
plt.axis('off')
plt.show()
```
As we can see, the generated images more look like art than realistic images. As the autoencoder was allowed to
structure the latent space in whichever way it suits the reconstruction best, there is no incentive to map every possible
latent vector to realistic images. Furthermore, the distribution in latent space is unknown to us and doesn’t necessarily
follow a multivariate normal distribution. Thus, we can conclude that vanilla autoencoders are indeed not generative.

### 5.11.2 Finding visually similar images

One application of autoencoders is to build an image-based search engine to retrieve visually similar images. This can
be done by representing all images as their latent dimensionality, and find the closest $K$ images in this domain. The
first step to such a search engine is to encode all images into $z$. In the following, we will use the training set as a search
corpus, and the test set as queries to the system.

(Warning: the following cells can be computationally heavy for a weak CPU-only system. If you do not have a strong
computer and are not on Google Colab, you might want to skip the execution of the following cells and rely on the
results shown in the filled notebook)

```python
# We use the following model throughout this section.
# If you want to try a different latent dimensionality, change it here!
model = model_dict[128]["model"]
```

```python
def embed_imgs(model, data_loader):
    # Encode all images in the data_loader using model, and return both images and encodings
    img_list, embed_list = [], []
    model.eval()
    for imgs, _ in tqdm(data_loader, desc="Encoding images", leave=False):
        with torch.no_grad():
            z = model.encoder(imgs.to(model.device))
            img_list.append(imgs)
            embed_list.append(z)
    return (torch.cat(img_list, dim=0), torch.cat(embed_list, dim=0))
```

(continues on next page)
After encoding all images, we just need to write a function that finds the closest $K$ images and returns (or plots) those:

```python
def find_similar_images(query_img, query_z, key_embeds, K=8):
    # Find closest K images. We use the euclidean distance here but other like cosine distance can also be used.
    dist = torch.cdist(query_z[None, :], key_embeds[1], p=2)
    dist = dist.squeeze(dim=0)
    dist, indices = torch.sort(dist)
    # Plot K closest images
    imgs_to_display = torch.cat([query_img[None], key_embeds[0][indices[:K]]], dim=0)
    grid = torchvision.utils.make_grid(imgs_to_display, nrow=K+1, normalize=True, range=(-1,1))
    grid = grid.permute(1, 2, 0)
    plt.figure(figsize=(12,3))
    plt.imshow(grid)
    plt.axis('off')
    plt.show()
```

```python
# Plot the closest images for the first N test images as example
for i in range(8):
    find_similar_images(test_img_embeds[0][i], test_img_embeds[1][i], key_embeds=train_img_embeds)
```
Based on our autoencoder, we see that we are able to retrieve many similar images to the test input. In particular, in row 4, we can spot that some test images might not be that different from the training set as we thought (same poster, just different scaling/color scaling). We also see that although we haven’t given the model any labels, it can cluster different classes in different parts of the latent space (airplane + ship, animals, etc.). This is why autoencoders can also be used as a pre-training strategy for deep networks, especially when we have a large set of unlabeled images (often the case). However, it should be noted that the background still plays a big role in autoencoders while it doesn’t for classification. Hence, we don’t get “perfect” clusters and need to finetune such models for classification.

Tensorboard clustering

Another way of exploring the similarity of images in the latent space is by dimensionality-reduction methods like PCA or T-SNE. Luckily, Tensorboard provides a nice interface for this and we can make use of it in the following:

```python
# We use the following model throughout this section.
# If you want to try a different latent dimensionality, change it here!
model = model_dict[128]["model"]
```

```python
# Create a summary writer
writer = SummaryWriter("tensorboard/")
writer.add_graph(model, torch.zeros(1,3,32,32, device=model.device)) # Optional: add graph of model
```

The function `add_embedding` allows us to add high-dimensional feature vectors to TensorBoard on which we can perform clustering. What we have to provide in the function are the feature vectors, additional metadata such as the labels, and the original images so that we can identify a specific image in the clustering.

```python
## In case you obtain the following error in the next cell, execute the import statements and last line in this cell
## AttributeError: module 'tensorflow._api.v2.io.gfile' has no attribute 'get_filesystem'
```

(continues on next page)
# import tensorflow as tf
# import tensorboard as tb
# tf.io.gfile = tb.compat.tensorflow_stub.io.gfile

[24]: # Note: the embedding projector in tensorboard is computationally heavy.
# Reduce the image amount below if your computer struggles with visualizing all 10k points
NUM_IMGS = len(test_set)

writer.add_embedding(test_img_embeds[1][:NUM_IMGS],  # Encodings per image
                      metadata=[test_set[i][1] for i in range(NUM_IMGS)],  # Adding the labels per image to the plot
                      label_img=(test_img_embeds[0][:NUM_IMGS]+1)/2.0)  # Adding the original images to the plot

Finally, we can run tensorboard to explore similarities among images:

[25]: %tensorboard --logdir tensorboard/

You should be able to see something similar as in the following image. In case the projector stays empty, try to start the TensorBoard outside of the Jupyter notebook.

![TensorBoard Screenshot](image)

Overall, we can see that the model indeed clustered images together that are visually similar. Especially the background color seems to be a crucial factor in the encoding. This correlates to the chosen loss function, here Mean Squared Error on pixel-level because the background is responsible for more than half of the pixels in an average image. Hence, the model learns to focus on it. Nevertheless, we can see that the encodings also separate a couple of classes in the latent space although it hasn’t seen any labels. This shows again that autoencoding can also be used as a “pre-training”/transfer learning task before classification.

[26]: # Closing the summary writer
writer.close()
5.11.3 Conclusion

In this tutorial, we have implemented our own autoencoder on small RGB images and explored various properties of the model. In contrast to variational autoencoders, vanilla AEs are not generative and can work on MSE loss functions. This makes them often easier to train. Both versions of AE can be used for dimensionality reduction, as we have seen for finding visually similar images beyond pixel distances. Despite autoencoders gaining less interest in the research community due to their more “theoretically” challenging counterpart of VAEs, autoencoders still find usage in a lot of applications like denoising and compression. Hence, AEs are an essential tool that every Deep Learning engineer/researcher should be familiar with.

5.12 Tutorial 10: Adversarial attacks

Filled notebook:

Pre-trained models and dataset:

In this tutorial, we will discuss adversarial attacks on deep image classification models. As we have seen in many of the previous tutorials so far, Deep Neural Networks are a very powerful tool to recognize patterns in data, and, for example, perform image classification on a human-level. However, we have not tested yet how robust these models actually are. Can we “trick” the model and find failure modes? Can we design images that the networks naturally classify incorrectly? Due to the high classification accuracy on unseen test data, we would expect that this can be difficult. However, in 2014, a research group at Google and NYU showed that deep CNNs can be easily fooled, just by adding some salient but carefully constructed noise to the images. For instance, take a look at the example below (figure credit - Goodfellow et al.):

The image on the left is the original image from ImageNet, and a deep CNN classifies the image correctly as “panda” with a class likelihood of 57%. Nevertheless, if we add a little noise to every pixel of the image, the prediction of the model changes completely. Instead of a panda, our CNN tells us that the image contains a “gibbon” with the confidence of over 99%. For a human, however, these two images look exactly alike, and you cannot distinguish which one has noise added and which doesn’t. While this first seems like a fun game to fool trained networks, it can have a serious impact on the usage of neural networks. More and more deep learning models are used in applications, such as for example autonomous driving. Imagine that someone who gains access to the camera input of the car, could make pedestrians “disappear” for the image understanding network by simply adding some noise to the input as shown below (the figure is taken from J.H. Metzen et al.). The first row shows the original image with the semantic segmentation output on the right (pedestrians red), while the second row shows the image with small noise and the corresponding segmentation prediction. The pedestrian becomes invisible for the network, and the car would think the road is clear ahead.
Some attack types don’t even require to add noise, but minor changes on a stop sign can be already sufficient for the network to recognize it as a “50km/h” speed sign (paper, paper). The consequences of such attacks can be devastating. Hence, every deep learning engineer who designs models for an application should be aware of the possibility of adversarial attacks.

To understand what makes CNNs vulnerable to such attacks, we will implement our own adversarial attack strategies in this notebook, and try to fool a deep neural network. Let’s being with importing our standard libraries:

```
### Standard libraries
import os
import json
import math
import time
import numpy as np
import scipy.linalg

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

### Progress bar
from tqdm.notebook import tqdm

### PyTorch
import torch
```

(continues on next page)
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim

# Torchvision
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms

# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError:
    # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install pytorch-lightning==1.0.3
import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial10"

# Setting the seed
pl.seed_everything(42)

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

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

Using device cuda:0

We have again a few download statements. This includes both a dataset, and a few pretrained patches we will use later.
5.12.1 Deep CNNs on ImageNet

For our experiments in this notebook, we will use common CNN architectures trained on the ImageNet dataset. Such models are luckily provided by PyTorch’s torchvision package, and hence we just need to load the model of our preference. For the results on the website and default on Google Colab, we use a ResNet34. Feel free to experiment with other architectures as well, the code is mainly independent of the specific architecture we choose.

```python
# Load CNN architecture pretrained on ImageNet
os.environ["TORCH_HOME"] = CHECKPOINT_PATH
pretrained_model = torchvision.models.resnet34(pretrained=True)
pretrained_model = pretrained_model.to(device)

# No gradients needed for the network
pretrained_model.eval()
for p in pretrained_model.parameters():
    p.requires_grad = False
```

To perform adversarial attacks, we also need a dataset to work on. Given that the CNN model has been trained on ImageNet, it is only fair to perform the attacks on data from ImageNet. For this, we provide a small set of pre-processed images from the original ImageNet dataset (note that this dataset is shared under the same license as the original ImageNet dataset). Specifically, we have 5 images for each of the 1000 labels of the dataset. We can load the data below, and create a corresponding data loader.

```python
# Mean and Std from ImageNet
NORM_MEAN = np.array([0.485, 0.456, 0.406])
NORM_STD = np.array([0.229, 0.224, 0.225])
# No resizing and center crop necessary as images are already preprocessed.
plain_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=NORM_MEAN,
                        std=NORM_STD)
])

# Load dataset and create data loader
imagenet_path = os.path.join(DATASET_PATH, "TinyImageNet/")
assert os.path.isdir(imagenet_path), "Could not find the ImageNet dataset at expected path " % imagenet_path + "Please make sure to have downloaded the ImageNet dataset here, or change the DATASET_PATH variable (currently set to $s)."
dataset = torchvision.datasets.ImageFolder(root=imagenet_path, transform=plain_transforms)
data_loader = data.DataLoader(dataset, batch_size=32, shuffle=False, drop_last=False, num_workers=8)
```

(continues on next page)
Before we start with our attacks, we should verify the performance of our model. As ImageNet has 1000 classes, simply looking at the accuracy is not sufficient to tell the performance of a model. Imagine a model that always predicts the true label as the second-highest class in its softmax output. Although we would say it recognizes the object in the image, it achieves an accuracy of 0. In ImageNet with 1000 classes, there is not always one clear label we can assign an image to. This is why for image classifications over so many classes, a common alternative metric is “Top-5 accuracy”, which tells us how many times the true label has been within the 5 most-likely predictions of the model. As models usually perform quite well on those, we report the error (1 - accuracy) instead of the accuracy:

```python
def eval_model(dataset_loader, img_func=None):
    tp, tp_5, counter = 0., 0., 0.
    for imgs, labels in tqdm(dataset_loader, desc="Validating...
    imgs = imgs.to(device)
    labels = labels.to(device)
    if img_func is not None:
        imgs = img_func(imgs, labels)
    with torch.no_grad():
        preds = pretrained_model(imgs)
    tp += (preds.argmax(dim=-1) == labels).sum()
    tp_5 += (preds.topk(5, dim=-1)[1] == labels[..., None]).any(dim=-1).sum()
    counter += preds.shape[0]
    acc = tp.float().item()/counter
    top5 = tp_5.float().item()/counter
    print("Top-1 error: \$4.2f%% \$ (100.0 * (1 - acc))")
    print("Top-5 error: \$4.2f%% \$ (100.0 * (1 - top5))")
    return acc, top5
```

[5]: eval_model(data_loader)

Top-1 error: 19.10%
Top-5 error: 4.30%

The ResNet34 achieves a decent error rate of 4.3% for the top-5 predictions. Next, we can look at some predictions of the model to get more familiar with the dataset. The function below plots an image along with a bar diagram of its predictions. We also prepare it to show adversarial examples for later applications.

```python
def show_prediction(img, label, pred, K=5, adv_img=None, noise=None):
    if isinstance(img, torch.Tensor):
        # Tensor image to numpy
        img = img.cpu().permute(1, 2, 0).numpy()
        img = (img * NORM_STD[None, None]) + NORM_MEAN[None, None]
        img = np.clip(img, a_min=0.0, a_max=1.0)
```

(continues on next page)
label = label.item()

# Plot on the left the image with the true label as title.
# On the right, have a horizontal bar plot with the top k predictions including probabilities
if noise is None or adv_img is None:
    fig, ax = plt.subplots(1, 2, figsize=(10,2), gridspec_kw={'width_ratios': [1, 1]})
else:
    fig, ax = plt.subplots(1, 5, figsize=(12,2), gridspec_kw={'width_ratios': [1, 1, 1, 1, 2]})

ax[0].imshow(img)
ax[0].set_title(label_names[label])
ax[0].axis('off')

if adv_img is not None and noise is not None:
    # Visualize adversarial images
    adv_img = adv_img.cpu().permute(1, 2, 0).numpy()
    adv_img = (adv_img * NORM_STD[None, None]) + NORM_MEAN[None, None]
    adv_img = np.clip(adv_img, a_min=0.0, a_max=1.0)
    ax[1].imshow(adv_img)
    ax[1].set_title('Adversarial')
    ax[1].axis('off')
    # Visualize noise
    noise = noise.cpu().permute(1, 2, 0).numpy()
    noise = noise * 0.5 + 0.5  # Scale between 0 to 1
    ax[2].imshow(noise)
    ax[2].set_title('Noise')
    ax[2].axis('off')
    # buffer
    ax[3].axis('off')

    if abs(pred.sum().item() - 1.0) > 1e-4:
        pred = torch.softmax(pred, dim=-1)
        topk_vals, topk_idx = pred.topk(K, dim=-1)
        topk_vals, topk_idx = topk_vals.cpu().numpy(), topk_idx.cpu().numpy()
        ax[-1].barh(np.arange(K), topk_vals*100.0, align='center', color=['C0' if topk_idx[i] != label else 'C2' for i in range(K)])
        ax[-1].set_yticks(np.arange(K))
        ax[-1].set_yticklabels([label_names[c] for c in topk_idx])
        ax[-1].invert_yaxis()
        ax[-1].set_xlabel('Confidence')
        ax[-1].set_title('Predictions')
        plt.show()
        plt.close()

Let's visualize a few images below:

```python
[8]: exmp_batch, label_batch = next(iter(data_loader))
with torch.no_grad():
    preds = pretrained_model(exmp_batch.to(device))
for i in range(1,17,5):
    show_prediction(exmp_batch[i], label_batch[i], preds[i])
```
The bar plot on the right shows the top-5 predictions of the model with their class probabilities. We denote the class probabilities with “confidence” as it somewhat resembles how confident the network is that the image is of one specific class. Some of the images have a highly peaked probability distribution, and we would expect the model to be rather robust against noise for those. However, we will see below that this is not always the case. Note that all of the images are of fish because the data loader doesn’t shuffle the dataset. Otherwise, we would get different images every time we run the notebook, which would make it hard to discuss the results on the static version.

5.12.2 White-box adversarial attacks

There have been proposed many possible adversarial attack strategies, which all share the same goal: alternate the data/image input only a little bit to have a great impact on the model’s prediction. Specifically, if we look at the ImageNet predictions above, how can we have to change the image of the goldfish so that the model does not recognize the goldfish anymore? At the same time, the label of the image should not change, in the sense that a human would still clearly classify it as a goldfish. This is the same objective that the generator network has in the Generative Adversarial Network framework: try to fool another network (discriminator) by changing its input.

Adversarial attacks are usually grouped into “white-box” and “black-box” attacks. White-box attacks assume that we have access to the model parameter and can, for example, calculate the gradients with respect to the input (similar as in GANs). Black-box attacks on the other hand have the harder task of not having any knowledge about the network, and can only obtain predictions for an image, but no gradients or the like. In this notebook, we will focus on white-box attacks as they are usually easier to implement and follow the intuition of Generative Adversarial Networks (GAN) as studied in lecture 10.

Fast Gradient Sign Method (FGSM)

One of the first attack strategies proposed is Fast Gradient Sign Method (FGSM), developed by Ian Goodfellow et al. in 2014. Given an image, we create an adversarial example by the following expression:

\[ \tilde{x} = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y)) \]

The term \( J(\theta, x, y) \) represents the loss of the network for classifying input image \( x \) as label \( y \); \( \epsilon \) is the intensity of the noise, and \( \tilde{x} \) the final adversarial example. The equation resembles SGD and is actually nothing else than that. We change the input image \( x \) in the direction of maximizing the loss \( J(\theta, x, y) \). This is exactly the other way round as during training, where we try to minimize the loss. The sign function and \( \epsilon \) can be seen as gradient clipping and learning rate specifically. We only allow our attack to change each pixel value by \( \epsilon \). You can also see that the attack can be performed very fast, as it only requires a single forward and backward pass. Let’s implement it below:
```
def fast_gradient_sign_method(model, imgs, labels, epsilon=0.02):
    # Determine prediction of the model
    inp_imgs = imgs.clone().requires_grad_()
    preds = model(inp_imgs.to(device))
    preds = F.log_softmax(preds, dim=-1)
    # Calculate loss by NLL
    loss = -torch.gather(preds, 1, labels.to(device).unsqueeze(dim=-1))
    loss.sum().backward()
    # Update image to adversarial example as written above
    noise_grad = torch.sign(inp_imgs.grad.to(imgs.device))
    fake_imgs = imgs + epsilon * noise_grad
    fake_imgs.detach_()
    return fake_imgs, noise_grad
```

The default value of $\epsilon = 0.02$ corresponds to changing a pixel value by about 1 in the range of 0 to 255, e.g. changing 127 to 128. This difference is marginal and can often not be recognized by humans. Let’s try it below on our example images:

```
adv_imgs, noise_grad = fast_gradient_sign_method(pretrained_model, exmp_batch, label_batch, epsilon=0.02)
with torch.no_grad():
    adv_preds = pretrained_model(adv_imgs.to(device))

for i in range(1, 17, 5):
    show_prediction(exmp_batch[i], label_batch[i], adv_preds[i], adv_img=adv_imgs[i], noise=noise_grad[i])
```
Despite the minor amount of noise, we are able to fool the network on all of our examples. None of the labels have made it into the top-5 for the four images, showing that we indeed fooled the model. We can also check the accuracy of the model on the adversarial images:

\[\text{Top-1 error: 93.56\%} \quad \text{Top-5 error: 60.52\%}\]

As expected, the model is fooled on almost every image at least for the top-1 error, and more than half don’t have the true label in their top-5. This is a quite significant difference compared to the error rate of 4.3\% on the clean images. However, note that the predictions remain semantically similar. For instance, in the images we visualized above, the tench is still recognized as another fish, as well as the great white shark being a dugong. FGSM could be adapted to increase the probability of a specific class instead of minimizing the probability of a label, but for those, there are usually better attacks such as the adversarial patch.

### Adversarial Patches

Instead of changing every pixel by a little bit, we can also try to change a small part of the image into whatever values we would like. In other words, we will create a small image patch that covers a minor part of the original image but causes the model to confidently predict a specific class we choose. This form of attack is an even bigger threat in real-world applications than FSGM. Imagine a network in an autonomous car that receives a live image from a camera. Another driver could print out a specific pattern and put it on the back of his/her vehicle to make the autonomous car believe that the car is actually a pedestrian. Meanwhile, humans would not notice it. Tom Brown et al. proposed a way of learning such adversarial image patches robustly in 2017 and provided a short demonstration on YouTube.

Interestingly, if you add a small picture of the target class (here toaster) to the original image, the model does not pick it up at all. A specifically designed patch, however, which only roughly looks like a toaster, can change the network’s prediction instantaneously.
Let’s take a closer look at how we can actually train such patches. The general idea is very similar to FSGM in the sense that we calculate gradients for the input, and update our adversarial input correspondingly. However, there are also some differences in the setup. Firstly, we do not calculate a gradient for every pixel. Instead, we replace parts of the input image with our patch and then calculate the gradients just for our patch. Secondly, we don’t just do it for one image, but we want the patch to work with any possible image. Hence, we have a whole training loop where we train the patch using SGD. Lastly, image patches are usually designed to make the model predict a specific class, not just any other arbitrary class except the true label. For instance, we can try to create a patch for the class “toaster” and train the patch so that our pretrained model predicts the class “toaster” for any image with the patch in it.

Additionally, to the setup described above, there are a couple of design choices we can take. For instance, Brown et al. randomly rotated and scaled the patch during training before placing it at a random position in an input image. This makes the patch more robust to small changes and is necessary if we want to fool a neural network in a real-world application. For simplicity, we will only focus on making the patch robust to the location in the image. Given a batch of input images and a patch, we can add the patch as follows:

```python
[12]: def place_patch(img, patch):
    for i in range(img.shape[0]):
        img[i,:,h_offset:h_offset+patch.shape[1],w_offset:w_offset+patch.shape[2]] = patch_forward(patch)
    return img
```

The patch itself will be an `nn.Parameter` whose values are in the range between $-\infty$ and $\infty$. Images are, however, naturally limited in their range, and thus we write a small function that maps the parameter into the image value range of ImageNet:

```python
[13]: TENSOR_MEANS, TENSOR_STD = torch.FloatTensor(NORM_MEAN)[:,:,None], torch.FloatTensor(NORM_STD)[:,:,None]
    def patch_forward(patch):
        # Map patch values from [-infty,infty] to ImageNet min and max
        patch = (torch.tanh(patch) + 1 - 2 * TENSOR_MEANS) / (2 * TENSOR_STD)
```

(continues on next page)
Before looking at the actual training code, we can write a small evaluation function. We evaluate the success of a patch by how many times we were able to fool the network into predicting our target class. A simple function for this is implemented below.

```python
[14]: def eval_patch(model, patch, val_loader, target_class):
    model.eval()
    tp, tp_5, counter = 0., 0., 0.
    with torch.no_grad():
        for img, img_labels in tqdm(val_loader, desc="Validating...", leave=False):
            # For stability, place the patch at 4 random locations per image, and average the performance
            for _ in range(4):
                patch_img = place_patch(img, patch)
                patch_img = patch_img.to(device)
                img_labels = img_labels.to(device)
                pred = model(patch_img)
                # In the accuracy calculation, we need to exclude the images that are of our target class
                # as we would not "fool" the model into predicting those
                tp += torch.logical_and(pred.argmax(dim=-1) == target_class, img_labels != target_class).sum()
                tp_5 += torch.logical_and((pred.topk(5, dim=-1)[1] == target_class).any(dim=-1), img_labels != target_class).sum()
                counter += (img_labels != target_class).sum()
    acc = tp/counter
    top5 = tp_5/counter
    return acc, top5
```

Finally, we can look at the training loop. Given a model to fool, a target class to design the patch for, and a size \( k \) of the patch in the number of pixels, we first start by creating a parameter of size \( 3 \times k \times k \). These are the only parameters we will train, and the network itself remains untouched. We use a simple SGD optimizer with momentum to minimize the classification loss of the model given the patch in the image. While we first start with a very high loss due to the good initial performance of the network, the loss quickly decreases once we start changing the patch. In the end, the patch will represent patterns that are characteristic of the class. For instance, if we would want the model to predict a “goldfish” in every image, we would expect the pattern to look somewhat like a goldfish. Over the iterations, the model finetunes the pattern and, hopefully, achieves a high fooling accuracy.

```python
[15]: def patch_attack(model, target_class, patch_size=64, num_epochs=5):
    # Leave a small set of images out to check generalization
    # In most of our experiments, the performance on the hold-out data points
    # was as good as on the training set. Overfitting was little possible due
    # to the small size of the patches.
    train_set, val_set = torch.utils.data.random_split(dataset, [4500, 500])
    train_loader = data.DataLoader(train_set, batch_size=32, shuffle=True, drop_last=True, num_workers=8)
    val_loader = data.DataLoader(val_set, batch_size=32, shuffle=False, drop_last=False, num_workers=4)

    # Create parameter and optimizer
    if not isinstance(patch_size, tuple):
        patch_size = (patch_size, patch_size)
    patch = nn.Parameter(torch.zeros(3, patch_size[0], patch_size[1]), requires_grad=True)
    optimizer = torch.optim.SGD([patch], lr=1e-1, momentum=0.8)
```

(continues on next page)
loss_module = nn.CrossEntropyLoss()

# Training loop
for epoch in range(num_epochs):
    t = tqdm(train_loader, leave=False)
    for img, _ in t:
        img = place_patch(img, patch)
        img = img.to(device)
        pred = model(img)
        labels = torch.zeros(img.shape[0], device=pred.device, dtype=torch.long).
        fill_(target_class)
        loss = loss_module(pred, labels)
        optimizer.zero_grad()
        loss.mean().backward()
        optimizer.step()
        t.set_description("Epoch %i, Loss: %.2f" % (epoch, loss.item()))

# Final validation
acc, top5 = eval_patch(model, patch, val_loader, target_class)
return patch.data, {"acc": acc.item(), "top5": top5.item()}

To get some experience with what to expect from an adversarial patch attack, we want to train multiple patches for different classes. As the training of a patch can take one or two minutes on a GPU, we have provided a couple of pre-trained patches including their results on the full dataset. The results are saved in a JSON file, which is loaded below.

[16]:
# Load evaluation results of the pretrained patches
json_results_file = os.path.join(CHECKPOINT_PATH, "patch_results.json")
json_results = {}
if os.path.isfile(json_results_file):
    with open(json_results_file, "r") as f:
        json_results = json.load(f)

# If you train new patches, you can save the results via calling this function
def save_results(patch_dict):
    result_dict = {cname: {psize: [t.item() if isinstance(t, torch.Tensor) else t for t in patch_dict[cname][psize]['results']] for psize in patch_dict[cname]} for cname in patch_dict}
    with open(os.path.join(CHECKPOINT_PATH, "patch_results.json"), "w") as f:
        json.dump(result_dict, f, indent=4)

Additionally, we implement a function to train and evaluate patches for a list of classes and patch sizes. The pretrained patches include the classes toaster, goldfish, school bus, lipstick, and pineapple. We chose the classes arbitrarily to cover multiple domains (animals, vehicles, fruits, devices, etc.). We trained each class for three different patch sizes: 32 × 32 pixels, 48 × 48 pixels, and 64 × 64 pixels. We can load them in the two cells below.

[17]:
def get_patches(class_names, patch_sizes):
    result_dict = dict()
    # Loop over all classes and patch sizes
    for name in class_names:
        result_dict[name] = dict()
        for patch_size in patch_sizes:
            c = label_names.index(name)
file_name = os.path.join(CHECKPOINT_PATH, "%s_%i_patch.pt" % (name, patch_size))

# Load patch if pretrained file exists, otherwise start training
if not os.path.isfile(file_name):
    patch, val_results = patch_attack(pretrained_model, target_class=c, patch_size=patch_size, num_epochs=5)
    print("Validation results for %s and %i:" % (name, patch_size), val_results)
    torch.save(patch, file_name)
else:
    patch = torch.load(file_name)

# Load evaluation results if exist, otherwise manually evaluate the patch
if name in json_results:
    results = json_results[name][str(patch_size)]
else:
    results = eval_patch(pretrained_model, patch, data_loader, target_class=c)

# Store results and the patches in a dict for better access
result_dict[name][patch_size] = {
    "results": results,
    "patch": patch
}

return result_dict

Feel free to add any additional classes and/or patch sizes.

[18]:
class_names = ["toaster", "goldfish", "school bus", "lipstick", "pineapple"]
patch_sizes = [32, 48, 64]

patch_dict = get_patches(class_names, patch_sizes)
# save_results(patch_dict) # Uncomment if you add new class names and want to save the new results

Before looking at the quantitative results, we can actually visualize the patches.

[19]:
def show_patches():
    fig, ax = plt.subplots(len(patch_sizes), len(class_names), figsize=(len(class_names)*2.2, len(patch_sizes)*2.2))
    for c_idx, cname in enumerate(class_names):
        for p_idx, psize in enumerate(patch_sizes):
            patch = patch_dict[cname][psize]["patch"]
            patch = (torch.tanh(patch) + 1) / 2 # Parameter to pixel values
            patch = patch.cpu().permute(1, 2, 0).numpy()
            patch = np.clip(patch, a_min=0.0, a_max=1.0)
            ax[p_idx][c_idx].imshow(patch)
            ax[p_idx][c_idx].set_title("%s, size %i" % (cname, psize))
            ax[p_idx][c_idx].axis('off')
    fig.subplots_adjust(hspace=0.3, wspace=0.3)
    plt.show()
show_patches()
We can see a clear difference between patches of different classes and sizes. In the smallest size, 32 × 32 pixels, some of the patches clearly resemble their class. For instance, the goldfish patch clearly shows a goldfish. The eye and the color are very characteristic of the class. Overall, the patches with 32 pixels have very strong colors that are typical for their class (yellow school bus, pink lipstick, greenish pineapple). The larger the patch becomes, the more stretched the pattern becomes. For the goldfish, we can still spot regions that might represent eyes and the characteristic orange color, but it is not clearly a single fish anymore. For the pineapple, we might interpret the top part of the image as the leaves of pineapple fruit, but it is more abstract than our small patches. Nevertheless, we can easily spot the alignment of the patch to class, even on the largest scale.

Let’s now look at the quantitative results.

```python
def show_table(top_1=True):
    i = 0 if top_1 else 1
    table = [[name] + ['%4.2f%%' % (100.0 * patch_dict[name][psize]['results'][i]) for psize in patch_sizes] for name in class_names]
    display(HTML(tabulate.tabulate(table, tablefmt='html', headers=['Class name'] + ['Patch size %ix%i' % (psize, psize) for psize in patch_sizes])))
```

226 Chapter 5. Current progress
First, we will create a table of top-1 accuracy, meaning that how many images have been classified with the target class as highest prediction?

```
[22]: show_table(top_1=True)
<IPython.core.display.HTML object>
```

The clear trend, that we would also have expected, is that the larger the patch, the easier it is to fool the model. For the largest patch size of 64 × 64, we are able to fool the model on almost all images, despite the patch covering only 8% of the image. The smallest patch actually covers 2% of the image, which is almost neglectable. Still, the fooling accuracy is quite remarkable. A large variation can be however seen across classes. While school bus and pineapple seem to be classes that were easily predicted, toaster and lipstick seem to be much harder for creating a patch. It is hard to intuitively explain why our patches underperform on those classes. Nonetheless, a fooling accuracy of >40% is still very good for such a tiny patch.

Let’s also take a look at the top-5 accuracy:

```
[23]: show_table(top_1=False)
<IPython.core.display.HTML object>
```

We see a very similar pattern across classes and patch sizes. The patch size 64 obtains >99.7% top-5 accuracy for any class, showing that we can almost fool the network on any image. A top-5 accuracy of >70% for the hard classes and small patches is still impressive and shows how vulnerable deep CNNs are to such attacks.

Finally, let’s create some example visualizations of the patch attack in action.

```
[24]: def perform_patch_attack(patch):
    patch_batch = exmp_batch.clone()
    patch_batch = place_patch(patch_batch, patch)
    with torch.no_grad():
        patch_preds = pretrained_model(patch_batch.to(device))
    for i in range(1,17,5):
        show_prediction(patch_batch[i], label_batch[i], patch_preds[i])

[25]: perform_patch_attack(patch_dict['goldfish'][32]['patch'])
```

![Example visualization of the patch attack in action](image)
The tiny goldfish patch can change all of the predictions to “goldfish” as top class. Note that the patch attacks work especially well if the input image is semantically similar to the target class (e.g. a fish and the target class “goldfish” works better than an airplane image with that patch). Nevertheless, we can also let the network predict semantically dis-similar classes by using a larger patch:

```python
[26]: perform_patch_attack(patch_dict['school bus'][64]['patch'])
```
Although none of the images have anything to do with an American school bus, the high confidence of often 100% shows how powerful such attacks can be. With a few lines of code and access to the model, we were able to generate patches that we add to any image to make the model predict any class we want.

**Transferability of white-box attacks**

FGSM and the adversarial patch attack were both focused on one specific image. However, can we transfer those attacks to other models? The adversarial patch attack as proposed in Brown et al., was originally trained on multiple models, and hence was also able to work on many different network architecture. But how different are the patches for different models anyway? For instance, let’s evaluate some of our patches trained above on a different network, e.g. DenseNet121.

```python
[27]: transfer_model = torchvision.models.densenet121(pretrained=True)
transfer_model = transfer_model.to(device)

# No gradients needed for the network
transfer_model.eval()
for p in transfer_model.parameters():
    p.requires_grad = False

Feel free to change the class name and/or patch size below to test out different patches.

[28]: class_name = 'pineapple'
patch_size = 64
print("Testing patch \"%s\" of size \$ix\$i" % (class_name, patch_size, patch_size))

results = eval_patch(transfer_model,
    patch_dict[class_name][patch_size]["patch"],
    data_loader,
    target_class=label_names.index(class_name))

print("Top-1 fool accuracy: \$4.2f%%\" % (results[0] * 100.0))
print("Top-5 fool accuracy: \$4.2f%%\" % (results[1] * 100.0))

Testing patch "pineapple" of size 64x64

Top-1 fool accuracy: 65.28%
Top-5 fool accuracy: 82.58%
```
Although the fool accuracy is significantly lower than on the original ResNet34, it still has a considerable impact on DenseNet although the networks have completely different architectures and weights. If you would compare more patches and models, some would work better than others. However, one aspect which allows patch attacks to generalize well is if all the networks have been trained on the same data. In this case, all networks have been trained on ImageNet. Dataset biases make the networks recognize specific patterns in the underlying image data that humans would not have seen, and/or only work for the given dataset. This is why the knowledge of what data has been used to train a specific model is already worth a lot in the context of adversarial attacks.

5.12.3 Protecting against adversarial attacks

There are many more attack strategies than just FGSM and adversarial patches that we haven’t discussed and implemented ourselves here. However, what about the other perspective? What can we do to protect a network against adversarial attacks? The sad truth to this is: not much.

White-box attacks require access to the model and its gradient calculation. The easiest way of preventing this is by ensuring safe, private storage of the model and its weights. However, some attacks, called black-box attacks, also work without access to the model’s parameters, or white-box attacks can also generalize as we have seen above on our short test on transferability.

So, how could we eventually protect a model? An intuitive approach would be to train/finetune a model on such adversarial images, leading to an adversarial training similar to a GAN. During training, we would pretend to be the attacker, and use for example FGSM as an augmentation strategy. However, this usually just ends up in an oscillation of the defending network between weak spots. Another common trick to increase robustness against adversarial attacks is defensive distillation (Papernot et al.). Instead of training the model on the dataset labels, we train a secondary model on the softmax predictions of the first one. This way, the loss surface is “smoothed” in the directions an attacker might try to exploit, and it becomes more difficult for the attacker to find adversarial examples. Nevertheless, there hasn’t been found the one true strategy that works against all possible adversarial attacks.

Why are CNNs, or neural networks in general, so vulnerable to adversarial attacks? While there are many possible explanations, the most intuitive is that neural networks don’t know what they don’t know. Even a large dataset represents just a few sparse points in the extremely large space of possible images. A lot of the input space has not been seen by the network during training, and hence, we cannot guarantee that the prediction for those images is any useful. The network instead learns a very good classification on a smaller region, often referred to as manifold, while ignoring the points outside of it. NNs with uncertainty prediction could potentially help to discover what the network does not know. Another possible explanation lies in the activation function. As we know, most CNNs use ReLU-based activation functions. While those have enabled great success in training deep neural networks due to their stable gradient for positive values, they also constitute a possible flaw. The output range of a ReLU neuron can be arbitrarily high. Thus, if we design a patch or the noise in the image to cause a very high value for a single neuron, it can overpower many other features in the network. Thus, although ReLU stabilizes training, it also offers a potential point of attack for adversaries.

5.12.4 Conclusion

In this tutorial, we have looked at different forms of adversarial attacks. Deep CNNs can be fooled by only slight modifications to the input. Whether it is a carefully designed noise pattern, unnoticeable for a human, or a small patch, we are able to manipulate the networks’ predictions significantly. The fact that even white-box attacks can be transferable across networks, and that there exist no suitable protections against all possible adversarial attacks, make this concept a massive problem for real-world applications. While adversarial attacks can also be used for improving/training a robust model or a GAN, it is not close to being solved yet. This is also because neural networks are currently complex, unknown non-linear functions in high-dimensional looking for correlations instead of causation. In the next years, we might hopefully see an improvement in the stability of such models by using causal approaches and/or introducing uncertainty.
5.13 Tutorial 11: Normalizing Flows for image modeling

Filled notebook:

Pre-trained models:

In this tutorial, we will take a closer look at complex, deep normalizing flows. The most popular, current application of deep normalizing flows is to model datasets of images. As for other generative models, images are a good domain to start working on because (1) CNNs are widely studied and strong models exist, (2) images are high-dimensional and complex, and (3) images are discrete integers. In this tutorial, we will review current advances in normalizing flows for image modeling, and get hands-on experience on coding normalizing flows. Note that normalizing flows are commonly parameter heavy and therefore computationally expensive. We will use relatively simple and shallow flows to save computational cost and allow you to run the notebook on CPU, but keep in mind that a simple way to improve the scores of the flows we study here is to make them deeper.

Throughout this notebook, we make use of PyTorch Lightning. The first cell imports our usual libraries.

```python
## Standard libraries
import os
import math
import time
import numpy as np
## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()
## Progress bar
from tqdm.notebook import tqdm
## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
```

(continues on next page)
import torch.optim as optim  
# Torchvision
import torchvision
from torchvision.datasets import MNIST
from torchvision import transforms  
# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError:  
    # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
    !pip install pytorch-lightning==1.0.3

import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial11"

# Setting the seed
pl.seed_everything(42)
# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.determinstic = True
torch.backends.cudnn.benchmark = False

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

Using device cuda:0

Again, we have a few pretrained models. We download them below to the specified path above.

[2]:
import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial11/"
# Files to download
pretrained_files = ["MNISTFlow_simple.ckpt", "MNISTFlow_vardeq.ckpt", "MNISTFlow_multiscale.ckpt"]
# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print("Downloading %s..." % file_url)
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive folder, or contact the author with the full output including the following error:", e)
We will use the MNIST dataset in this notebook. MNIST constitutes, despite its simplicity, a challenge for small generative models as it requires the global understanding of an image. At the same time, we can easily judge whether generated images come from the same distribution as the dataset (i.e. represent real digits), or not.

To deal better with the discrete nature of the images, we transform them from a range of 0-1 to a range of 0-255 as integers.

```python
# Convert images from 0-1 to 0-255 (integers)
def discretize(sample):
    return (sample * 255).to(torch.int32)

# Transformations applied on each image => make them a tensor and discretize
transform = transforms.Compose([transforms.ToTensor(),
                                 discretize])

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = MNIST(root=DATASET_PATH, train=True, transform=transform, download=True)
pl.seed_everything(42)
train_set, val_set = torch.utils.data.random_split(train_dataset, [50000, 10000])

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

# We define a set of data loaders that we can use for various purposes later.
# Note that for actually training a model, we will use different data loaders
# with a lower batch size.
train_loader = data.DataLoader(train_set, batch_size=256, shuffle=False, drop_last=False)
val_loader = data.DataLoader(val_set, batch_size=64, shuffle=False, drop_last=False, num_workers=4)
test_loader = data.DataLoader(test_set, batch_size=64, shuffle=False, drop_last=False, num_workers=4)
```

In addition, we will define below a function to simplify the visualization of images/samples. Some training examples of the MNIST dataset is shown below.

```python
# Form a grid of pictures (we use max. 8 columns)
num_imgs = imgs.shape[0]
if isinstance(imgs, torch.Tensor):
is_int = imgs.dtype==torch.int32
else:
is_int = len(imgs)

nrow = min(num_imgs, row_size)
col = int(math.ceil(num_imgs/nrow))
imgs = torchvision.utils.make_grid(imgs, nrow=nrow, pad_value=128 if is_int else 0.5)

# Plot the grid
plt.figure(figsize=(1.5*nrow, 1.5*ncol))
plt.imshow(np.transpose(np_imgs, (1,2,0)), interpolation='nearest')
plt.axis('off')
if title is not None:
    plt.title(title)
plt.show()
plt.close()

show_imgs([train_set[i][0] for i in range(8)])
```

234 Chapter 5. Current progress
5.13.1 Normalizing Flows as generative model

In the previous lectures, we have seen Energy-based models, Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs) as example of generative models. However, none of them explicitly learn the probability density function \( p(x) \) of the real input data. While VAEs model a lower bound, energy-based models only implicitly learn the probability density. GANs on the other hand provide us a sampling mechanism for generating new data, without offering a likelihood estimate. The generative model we will look at here, called Normalizing Flows, actually models the true data distribution \( p(x) \) and provides us with an exact likelihood estimate. Below, we can visually compare VAEs, GANs and Flows (figure credit - Lilian Weng):

**GAN**: minimax the classification error loss.

**VAE**: maximize ELBO.

**Flow-based generative models**: minimize the negative log-likelihood

The major difference compared to VAEs is that flows use *invertible* functions \( f \) to map the input data \( x \) to a latent representation \( z \). To realize this, \( z \) must be of the same shape as \( x \). This is in contrast to VAEs where \( z \) is usually much lower dimensional than the original input data. However, an invertible mapping also means that for every data point \( x \), we have a corresponding latent representation \( z \) which allows us to perform lossless reconstruction (\( z \) to \( x \)). In the visualization above, this means that \( x = x' \) for flows, no matter what invertible function \( f \) and input \( x \) we choose.
Nonetheless, how are normalizing flows modeling a probability density with an invertible function? The answer to this question is the rule for change of variables. Specifically, given a prior density $p_z(z)$ (e.g. Gaussian) and an invertible function $f$, we can determine $p_x(x)$ as follows:

\[
\int p_x(x)dx = \int p_z(z)dz = 1 \quad \text{(by definition of a probability distribution)}
\]

\[
\Leftrightarrow p_x(x) = p_z(z) \left| \frac{dz}{dx} \right| = p_z(f(x)) \left| \frac{df(x)}{dx} \right|
\]

Hence, in order to determine the probability of $x$, we only need to determine its probability in latent space, and get the derivative of $f$. Note that this is for a univariate distribution, and $f$ is required to be invertible and smooth. For a multivariate case, the derivative becomes a Jacobian of which we need to take the determinant. As we usually use the log-likelihood as objective, we write the multivariate term with logarithms below:

\[
\log p_x(x) = \log p_z(f(x)) + \log \left| \det \frac{df(x)}{dx} \right|
\]

Although we now know how a normalizing flow obtains its likelihood, it might not be clear what a normalizing flow does intuitively. For this, we should look from the inverse perspective of the flow starting with the prior probability density $p_z(z)$. If we apply an invertible function on it, we effectively “transform” its probability density. For instance, if $f^{-1}(z) = z+1$, we shift the density by one while still remaining a valid probability distribution, and being invertible. We can also apply more complex transformations, like scaling:

\[
f^{-1}(z) = 2z + 1
\]

Although we now know how a normalizing flow obtains its likelihood, it might not be clear what a normalizing flow does intuitively. For this, we should look from the inverse perspective of the flow starting with the prior probability density $p_z(z)$. If we apply an invertible function on it, we effectively “transform” its probability density. For instance, if $f^{-1}(z) = z+1$, we shift the density by one while still remaining a valid probability distribution, and being invertible. We can also apply more complex transformations, like scaling: $f^{-1}(z) = 2z + 1$, but there you might see a difference. When you scale, you also change the volume of the probability density, as for example on uniform distributions (figure credit - Eric Jang):

\[
f : \mathbb{R} \rightarrow \mathbb{R}, f(x) = 2x + 1
\]

You can see that the height of $p(y)$ should be lower than $p(x)$ after scaling. This change in volume represents $\left| \frac{df(x)}{dx} \right|$ in our equation above, and ensures that even after scaling, we still have a valid probability distribution. We can go on with making our function $f$ more complex. However, the more complex $f$ becomes, the harder it will be to find the inverse $f^{-1}$ of it, and to calculate the log-determinant of the Jacobian $\log \left| \det \frac{df(x)}{dx} \right|$. An easier trick to stack multiple invertible functions $f_1, \ldots, f_K$ after each other, as all together, they still represent a single, invertible function. Using multiple, learnable invertible functions, a normalizing flow attempts to transform $p_z(z)$ slowly into a more complex distribution which should finally be $p_x(x)$. We visualize the idea below (figure credit - Lilian Weng):
Starting from \(z_0\), which follows the prior Gaussian distribution, we sequentially apply the invertible functions \(f_1, f_2, \ldots, f_K\), until \(z_K\) represents \(x\). Note that in the figure above, the functions \(f\) represent the inverted function from \(f\) we had above (here: \(f: Z \rightarrow X\), above: \(f: X \rightarrow Z\)). This is just a different notation and has no impact on the actual flow design because all \(f\) need to be invertible anyways. When we estimate the log likelihood of a data point \(x\) as in the equations above, we run the flows in the opposite direction than visualized above. Multiple flow layers have been proposed that use a neural network as learnable parameters, such as the planar and radial flow. However, we will focus here on flows that are commonly used in image modeling, and will discuss them in the rest of the notebook along with the details of how to train a normalizing flow.

5.13.2 Normalizing Flows on images

To become familiar with normalizing flows, especially for the application of image modeling, it is best to discuss the different elements in a flow along with the implementation. As a general concept, we want to build a normalizing flow that maps an input image (here MNIST) to an equally sized latent space:

As a first step, we will implement a template of a normalizing flow in PyTorch Lightning. During training and validation, a normalizing flow performs density estimation in the forward direction. For this, we apply a series of flow transformations on the input \(x\) and estimate the probability of the input by determining the probability of the transformed point \(z\) given a prior, and the change of volume caused by the transformations. During inference, we can do both density estimation and sampling new points by inverting the flow transformations. Therefore, we define a function _get_likelihood which performs density estimation, and sample to generate new examples. The functions training_step, validation_step and test_step all make use of _get_likelihood.

The standard metric used in generative models, and in particular normalizing flows, is bits per dimensions (bpd). Bpd is motivated from an information theory perspective and describes how many bits we would need to encode a particular example in our modeled distribution. The less bits we need, the more likely the example in our distribution. When we test for the bits per dimension of our test dataset, we can judge whether our model generalizes to new samples of the dataset and didn’t memorize the training dataset. In order to calculate the bits per dimension score, we can rely on the negative log-likelihood and change the log base (as bits are binary while NLL is usually exponential):

\[
bpd = \text{nll} \cdot \log_2 (\exp(1)) \cdot \left(\prod d_i\right)^{-1}
\]

where \(d_1, \ldots, d_K\) are the dimensions of the input. For images, this would be the height, width and channel number. We divide the log likelihood by these extra dimensions to have a metric which we can compare for different image resolutions. In the original image space, MNIST examples have a bits per dimension score of 8 (we need 8 bits to encode each pixel as there are 256 possible values).

```python
[5]: class ImageFlow(pl.LightningModule):
    def __init__(self, flows, import_samples=8):
    (continues on next page)
```

5.13. Tutorial 11: Normalizing Flows for image modeling
""
Inputs:
flows - A list of flows (each a nn.Module) that should be applied on the
→ images.
import_samples - Number of importance samples to use during testing (see
→ explanation below). Can be changed at any time
""
super().__init__()
self.flows = nn.ModuleList(flows)
self.import_samples = import_samples
# Create prior distribution for final latent space
self.prior = torch.distributions.normal.Normal(loc=0.0, scale=1.0)
# Example input for visualizing the graph
self.example_input_array = train_set[0][0].unsqueeze(dim=0)

def forward(self, imgs):
    # The forward function is only used for visualizing the graph
    return self._get_likelihood(imgs)

def encode(self, imgs):
    # Given a batch of images, return the latent representation z and ldj of the
→ transformations
    z, ldj = imgs, torch.zeros(imgs.shape[0], device=self.device)
    for flow in self.flows:
        z, ldj = flow(z, ldj, reverse=False)
    return z, ldj

def _get_likelihood(self, imgs, return_ll=False):
    ""
    Given a batch of images, return the likelihood of those.
    If return_ll is True, this function returns the log likelihood of the input.
    Otherwise, the output metric is bits per dimension (scaled negative log
→ likelihood)
    ""
    z, ldj = self.encode(imgs)
    log_pz = self.prior.log_prob(z).sum(dim=[1,2,3])
    log_px = ldj + log_pz
    nll = -log_px
    # Calculating bits per dimension
    bpd = nll * np.log2(np.exp(1)) / np.prod(imgs.shape[1:])
    return bpd.mean() if not return_ll else log_px

@torch.no_grad()
def sample(self, img_shape, z_init=None):
    ""
    Sample a batch of images from the flow.
    ""
    # Sample latent representation from prior
    if z_init is None:
        z = self.prior.sample(sample_shape=img_shape).to(device)
    else:
        z = z_init.to(device)
(continues on next page)
# Transform z to x by inverting the flows

```python
ldj = torch.zeros(img_shape[0], device=device)
for flow in reversed(self.flows):
    z, ldj = flow(z, ldj, reverse=True)
return z
```

```python
def configure_optimizers(self):
    optimizer = optim.Adam(self.parameters(), lr=1e-3)
    # An scheduler is optional, but can help in flows to get the last bpd improvement.
    scheduler = optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.99)
    return [optimizer], [scheduler]
```

```python
def training_step(self, batch, batch_idx):
    # Normalizing flows are trained by maximum likelihood => return bpd
    loss = self._get_likelihood(batch[0])
    self.log('train_bpd', loss)
    return loss
```

```python
def validation_step(self, batch, batch_idx):
    loss = self._get_likelihood(batch[0])
    self.log('val_bpd', loss)
```

```python
def test_step(self, batch, batch_idx):
    # Perform importance sampling during testing => estimate likelihood M times for each image.
    samples = []
    for _ in range(self.import_samples):
        img_ll = self._get_likelihood(batch[0], return_ll=True)
        samples.append(img_ll)
        img_ll = torch.stack(samples, dim=-1)

    # To average the probabilities, we need to go from log-space to exp, and back to log.
    # Logsumexp provides us a stable implementation for this
    img_ll = torch.logsumexp(img_ll, dim=-1) - np.log(self.import_samples)

    # Calculate final bpd
    bpd = -img_ll * np.log2(np.exp(1)) / np.prod(batch[0].shape[1:])
    bpd = bpd.mean()
    self.log('test_bpd', bpd)
```

The `test_step` function differs from the training and validation step in that it makes use of importance sampling. We will discuss the motivation and details behind this after understanding how flows model discrete images in continuous space.
Dequantization

Normalizing flows rely on the rule of change of variables, which is naturally defined in continuous space. Applying flows directly on discrete data leads to undesired density models where arbitrarily high likelihood are placed on a few, particular values. See the illustration below:

The black points represent the discrete points, and the green volume the density modeled by a normalizing flow in continuous space. The flow would continue to increase the likelihood for \( x = 0, 1, 2, 3 \) while having no volume on any other point. Remember that in continuous space, we have the constraint that the overall volume of the probability density must be 1 (\( \int p(x)dx = 1 \)). Otherwise, we don’t model a probability distribution anymore. However, the discrete points \( x = 0, 1, 2, 3 \) represent delta peaks with no width in continuous space. This is why the flow can place an infinite high likelihood on these few points while still representing a distribution in continuous space. Nonetheless, the learned density does not tell us anything about the distribution among the discrete points, as in discrete space, the likelihoods of those four points would have to sum to 1, not to infinity.

To prevent such degenerated solutions, a common solution is to add a small amount of noise to each discrete value, which is also referred to as dequantization. Considering \( x \) as an integer (as it is the case for images), the dequantized representation \( v \) can be formulated as \( v = x + u \) where \( u \in [0, 1)^D \). Thus, the discrete value 1 is modeled by a distribution over the interval \([1.0, 2.0)\), the value 2 by an volume over \([2.0, 3.0)\), etc. Our objective of modeling \( p(x) \) becomes:

\[
p(x) = \int p(x + u)du = \int \frac{q(u|x)}{q(u|x)} p(x + u)du = \mathbb{E}_{u \sim q(u|x)} \left[ \frac{p(x + u)}{q(u|x)} \right]
\]

with \( q(u|x) \) being the noise distribution. For now, we assume it to be uniform, which can also be written as \( p(x) = \mathbb{E}_{u \sim U(0,1)^D} [p(x + u)] \).

In the following, we will implement Dequantization as a flow transformation itself. After adding noise to the discrete values, we additionally transform the volume into a Gaussian-like shape. This is done by scaling \( x + u \) between 0 and 1, and applying the inverse of the sigmoid function \( \sigma(z)^{-1} = \log z - \log 1 - z \). If we would not do this, we would face two problems:

1. The input is scaled between 0 and 256 while the prior distribution is a Gaussian with mean 0 and standard deviation 1. In the first iterations after initializing the parameters of the flow, we would have extremely low likelihoods for large values like 256. This would cause the training to diverge instantaneously.

2. As the output distribution is a Gaussian, it is beneficial for the flow to have a similarly shaped input distribution. This will reduce the modeling complexity that is required by the flow.

Overall, we can implement dequantization as follows:

```python
[6]: class Dequantization(nn.Module):
    def __init__(self, alpha=1e-5, quants=256):
        """
        Inputs:
        alpha - small constant that is used to scale the original input.
        Prevents dealing with values very close to 0 and 1 when inverting the sigmoid
        quants - Number of possible discrete values (usually 256 for 8-bit image)
        """
        super().__init__()
        self.alpha = alpha
        self.quants = quants

    def forward(self, z, ldj, reverse=False):
        if not reverse:
```

(continues on next page)
z, ldj = self.dequant(z, ldj)
z, ldj = self.sigmoid(z, ldj, reverse=True)
else:
z, ldj = self.sigmoid(z, ldj, reverse=False)
    z = z * self.quants
ldj += np.log(self.quants) * np.prod(z.shape[1:])
z = torch.floor(z).clamp(min=0, max=self.quants-1).to(torch.int32)
return z, ldj

def sigmoid(self, z, ldj, reverse=False):
    # Applies an invertible sigmoid transformation
    if not reverse:
        ldj += (-z-2*F.softplus(-z)).sum(dim=[1,2,3])
z = torch.sigmoid(z)
    else:
        z = z * (1 - self.alpha) + 0.5 * self.alpha  # Scale to prevent boundaries 0 and 1
        ldj += np.log(1 - self.alpha) * np.prod(z.shape[1:])
        ldj += (-torch.log(z) - torch.log(1-z)).sum(dim=[1,2,3])
z = torch.log(z) - torch.log(1-z)
return z, ldj
def dequant(self, z, ldj):
    # Transform discrete values to continuous volumes
z = z.to(torch.float32)
z = z + torch.rand_like(z).detach()
z = z / self.quants
ldj -= np.log(self.quants) * np.prod(z.shape[1:])
return z, ldj

A good check whether a flow is correctly implemented or not, is to verify that it is invertible. Hence, we will dequantize a randomly chosen training image, and then quantize it again. We would expect that we would get the exact same image out:

```python
[7]: # Testing invertibility of dequantization layer
pl.seed_everything(42)
orig_img = train_set[0][0].unsqueeze(dim=0)
ldj = torch.zeros(1,)
dequant_module = Dequantization()
deq_img, ldj = dequant_module(orig_img, ldj, reverse=False)
reconst_img, ldj = dequant_module(deq_img, ldj, reverse=True)
d1, d2 = torch.where(orig_img.squeeze() != reconst_img.squeeze())
if len(d1) != 0:
    print("Dequantization was not invertible.")
    for i in range(d1.shape[0]):
        print("Original value: ", orig_img[0,0,d1[i]], d2[i]).item())
        print("Reconstructed value: ", reconst_img[0,0,d1[i]], d2[i]).item())
else:
    print("Successfully inverted dequantization")

# Layer is not strictly invertible due to float precision constraints
# assert (orig_img == reconst_img).all().item()
Dequantization was not invertible.
Original value: 0
Reconstructed value: 1
```
In contrast to our expectation, the test fails. However, this is no reason to doubt our implementation here as only one single value is not equal to the original. This is caused due to numerical inaccuracies in the sigmoid invert. While the input space to the inverted sigmoid is scaled between 0 and 1, the output space is between $-\infty$ and $\infty$. And as we use 32 bits to represent the numbers (in addition to applying logs over and over again), such inaccuracies can occur and should not be worrisome. Nevertheless, it is good to be aware of them, and can be improved by using a double tensor (float64).

Finally, we can take our dequantization and actually visualize the distribution it transforms the discrete values into:

```python
[8]: def visualize_dequantization(quants, prior=None):
    
    """
    Function for visualizing the dequantization values of discrete values in continuous space
    """
    # Prior over discrete values. If not given, a uniform is assumed
    if prior is None:
        prior = np.ones(quants, dtype=np.float32) / quants
        prior = prior / prior.sum() * quants  # In the following, we assume 1 for each value means uniform distribution
    inp = torch.arange(-4, 4, 0.01).view(-1, 1, 1, 1)  # Possible continuous values we want to consider
    ldj = torch.zeros(inp.shape[0])
    dequant_module = Dequantization(quants=quants)
    # Invert dequantization on continuous values to find corresponding discrete value
    out, ldj = dequant_module.forward(inp, ldj, reverse=True)
    inp, out, prob = inp.squeeze().numpy(), out.squeeze().numpy(), ldj.exp().numpy()
    prob = prob * prior[out]  # Probability scaled by categorical prior
    # Plot volumes and continuous distribution
    sns.set_style("white")
    fig = plt.figure(figsize=(6,3))
    x_ticks = []
    for v in np.unique(out):
        indices = np.where(out==v)
        color = to_rgb("C%i"%v)
        plt.fill_between(inp[indices], prob[indices], np.zeros(indices[0].shape[0]), color=color+(0.5,), label=str(v))
        plt.plot([inp[indices[0][0]]]*2, [0, prob[indices[0][0]]], color=color)
        plt.plot([inp[indices[-1]]]*2, [0, prob[indices[-1]]], color=color)
        x_ticks.append(inp[indices[0][0]])
        x_ticks.append(inp.max())
    plt.xticks(x_ticks, ["%.1f"%x for x in x_ticks])
    plt.plot(inp,prob, color=(0.0,0.0,0.0))
    # Set final plot properties
    plt.ylim(0, prob.max()*1.1)
    plt.xlim(inp.min(), inp.max())
    plt.xlabel("z")
    plt.ylabel("Probability")
    plt.title("Dequantization distribution for %i discrete values" % quants)
    plt.legend()
    plt.show()
    plt.close()

visualize_dequantization(quants=8)
```
The visualized distribution shows the sub-volumes that are assigned to the different discrete values. The value 0 has its volume between $(-\infty, -1.9)$, the value 1 is represented by the interval $(-1.9, -1.1)$, etc. The volume for each discrete value has the same probability mass. That’s why the volumes close to the center (e.g. 3 and 4) have a smaller area on the z-axis as others ($z$ is being used to denote the output of the whole dequantization flow).

Effectively, the consecutive normalizing flow models discrete images by the following objective:

$$\log p(x) = \log \mathbb{E}_{u \sim q(u|x)} \left[ \frac{p(x + u)}{q(u|x)} \right] \geq \mathbb{E}_u \left[ \log \frac{p(x + u)}{q(u|x)} \right]$$

Although normalizing flows are exact in likelihood, we have a lower bound. Specifically, this is an example of the Jensen inequality because we need to move the log into the expectation so we can use Monte-carlo estimates. In general, this bound is considerably smaller than the ELBO in variational autoencoders. Actually, we can reduce the bound ourselves by estimating the expectation not by one, but by $M$ samples. In other words, we can apply importance sampling which leads to the following inequality:

$$\log p(x) = \log \mathbb{E}_{u \sim q(u|x)} \left[ \frac{p(x + u)}{q(u|x)} \right] \geq \mathbb{E}_u \left[ \log \frac{1}{M} \sum_{m=1}^{M} \frac{p(x + u_m)}{q(u_m|x)} \right] \geq \mathbb{E}_u \left[ \log \frac{p(x + u)}{q(u|x)} \right]$$

The importance sampling $\frac{1}{M} \sum_{m=1}^{M} \frac{p(x + u_m)}{q(u_m|x)}$ becomes $\mathbb{E}_{u \sim q(u|x)} \left[ \frac{p(x + u)}{q(u|x)} \right]$ if $M \to \infty$, so that the more samples we use, the tighter the bound is. During testing, we can make use of this property and have it implemented in `test_step` in `ImageFlow`. In theory, we could also use this tighter bound during training. However, related work has shown that this does not necessarily lead to an improvement given the additional computational cost, and it is more efficient to stick with a single estimate [5].

**Variational Dequantization**

Dequantization uses a uniform distribution for the noise $u$ which effectively leads to images being represented as hypercubes (cube in high dimensions) with sharp borders. However, modeling such sharp borders is not easy for a flow as it uses smooth transformations to convert it into a Gaussian distribution.

Another way of looking at it is if we change the prior distribution in the previous visualization. Imagine we have independent Gaussian noise on pixels which is commonly the case for any real-world taken picture. Therefore, the flow would have to model a distribution as above, but with the individual volumes scaled as follows:
Transforming such a probability into a Gaussian is a difficult task, especially with such hard borders. Dequantization has therefore been extended to more sophisticated, learnable distributions beyond uniform in a variational framework. In particular, if we remember the learning objective \( \log p(x) = \log \mathbb{E}_u \left[ \frac{p(x+u)}{q(u|x)} \right] \), the uniform distribution can be replaced by a learned distribution \( q_\theta(u|x) \) with support over \( u \in [0,1)^D \). This approach is called Variational Dequantization and has been proposed by Ho et al. [3]. How can we learn such a distribution? We can use a second normalizing flow that takes \( x \) as external input and learns a flexible distribution over \( u \). To ensure a support over \( [0,1)^D \), we can apply a sigmoid activation function as final flow transformation.

Inheriting the original dequantization class, we can implement variational dequantization as follows:

```python
[10]: class VariationalDequantization(Dequantization):
    def __init__(self, var_flows, alpha=1e-5):
        # Inputs:
        # var_flows - A list of flow transformations to use for modeling q(u|x)
        # alpha - Small constant, see Dequantization for details
        super().__init__(alpha=alpha)
        self.flows = nn.ModuleList(var_flows)

    def dequant(self, z, ldj):
        z = z.to(torch.float32)
        img = (z / 255.0) * 2 - 1 # We condition the flows on x, i.e. the original image
        # Prior of u is a uniform distribution as before
        # As most flow transformations are defined on [-infinity,+infinity], we apply an inverse sigmoid first.
        deq_noise = torch.rand_like(z).detach()
        deq_noise, ldj = self.sigmoid(deq_noise, ldj, reverse=True)
        for flow in self.flows:
            deq_noise, ldj = flow(deq_noise, ldj, reverse=False, orig_img=img)
```

(continues on next page)
deq_noise, ldj = self.sigmoid(deq_noise, ldj, reverse=False)

# After the flows, apply u as in standard dequantization
z = (z + deq_noise) / 256.0
ldj -= np.log(256.0) * np.prod(z.shape[1:])
return z, ldj

Variational dequantization can be used as a substitute for dequantization. We will compare dequantization and variational dequantization in later experiments.

**Coupling layers**

Next, we look at possible transformations to apply inside the flow. A recent popular flow layer, which works well in combination with deep neural networks, is the coupling layer introduced by Dinh et al. [1]. The input \( z \) is arbitrarily split into two parts, \( z_{1:j} \) and \( z_{j+1:d} \), of which the first remains unchanged by the flow. Yet, \( z_{1:j} \) is used to parameterize the transformation for the second part, \( z_{j+1:d} \). Various transformations have been proposed in recent time [3, 4], but here we will settle for the simplest and most efficient one: affine coupling. In this coupling layer, we apply an affine transformation by shifting the input by a bias \( \mu \) and scale it by \( \sigma \). In other words, our transformation looks as follows:

\[
\begin{align*}
z'_{j+1:d} &= \mu(z_{1:j}) + \sigma(z_{1:j}) \odot z_{j+1:d}
\end{align*}
\]

The functions \( \mu \) and \( \sigma \) are implemented as a shared neural network, and the sum and multiplication are performed element-wise. The LDJ is thereby the sum of the logs of the scaling factors: \( \sum_i \log \sigma_i(z_{1:j}) \). Inverting the layer can as simply be done as subtracting the bias and dividing by the scale:

\[
\begin{align*}
z_{j+1:d} &= (z'_{j+1:d} - \mu(z_{1:j})) / \sigma(z_{1:j})
\end{align*}
\]

We can also visualize the coupling layer in form of a computation graph, where \( z_1 \) represents \( z_{1:j} \), and \( z_2 \) represents \( z_{j+1:d} \).

In our implementation, we will realize the splitting of variables as masking. The variables to be transformed, \( z_{j+1:d} \), are masked when passing \( z \) to the shared network to predict the transformation parameters. When applying the transformation, we mask the parameters for \( z_{1:j} \) so that we have an identity operation for those variables:

```python
[11]: class CouplingLayer(nn.Module):
    def __init__(self, network, mask, c_in):
        """Coupling layer inside a normalizing flow.
        Inputs:
        network - A PyTorch nn.Module constituting the deep neural network for mu and sigma.
        mask - Binary mask (0 or 1) where 0 denotes that the element should be transformed, while 1 means the latent will be used as input to the NN.
        c_in - Number of input channels
        """
        super().__init__()
        self.network = network
        self.scaling_factor = nn.Parameter(torch.zeros(c_in))
        # Register mask as buffer as it is a tensor which is not a parameter,
        # but should be part of the modules state.
```

(continues on next page)
self.register_buffer('mask', mask)

def forward(self, z, ldj, reverse=False, orig_img=None):
    """
    Inputs:
    z - Latent input to the flow
    ldj - The current ldj of the previous flows.
    reverse - If True, we apply the inverse of the layer.
    orig_img (optional) - Only needed in VarDeq. Allows external
    input to condition the flow on (e.g. original image)
    """
    # Apply network to masked input
    z_in = z * self.mask
    if orig_img is None:
        nn_out = self.network(z_in)
    else:
        nn_out = self.network(torch.cat([z_in, orig_img], dim=1))
    s, t = nn_out.chunk(2, dim=1)
    # Stabilize scaling output
    s_fac = self.scaling_factor.exp().view(1, -1, 1, 1)
    s = torch.tanh(s / s_fac) * s_fac
    t = t * (1 - self.mask)
    # Affine transformation
    if not reverse:
        # Whether we first shift and then scale, or the other way round,
        # is a design choice, and usually does not have a big impact
        z = (z + t) * torch.exp(s)
        ldj += s.sum(dim=[1,2,3])
    else:
        z = (z * torch.exp(-s)) - t
        ldj -= s.sum(dim=[1,2,3])
    return z, ldj

For stabilization purposes, we apply a tanh activation function on the scaling output. This prevents sudden large output values for the scaling that can destabilize training. To still allow scaling factors smaller or larger than -1 and 1 respectively, we have a learnable parameter per dimension, called scaling_factor. This scales the tanh to different limits. Below, we visualize the effect of the scaling factor on the output activation of the scaling terms:

```
[12]: with torch.no_grad():
    x = torch.arange(-5,5,0.01)
    scaling_factors = [0.5, 1, 2]
    sns.set()
    fig, ax = plt.subplots(1, 3, figsize=(12,3))
    for i, scale in enumerate(scaling_factors):
        y = torch.tanh(x / scale) * scale
        ax[i].plot(x.numpy(), y.numpy())
        ax[i].set_title("Scaling factor: " + str(scale))
        ax[i].set_ylim(-3, 3)
    plt.subplots_adjust(wspace=0.4)
```
Coupling layers generalize to any masking technique we could think of. However, the most common approach for images is to split the input $z$ in half, using a checkerboard mask or channel mask. A checkerboard mask splits the variables across the height and width dimensions and assigns each other pixel to $z_{j+1:d}$. Thereby, the mask is shared across channels. In contrast, the channel mask assigns half of the channels to $z_{j+1:d}$, and the other half to $z_{1:j+1}$. Note that when we apply multiple coupling layers, we invert the masking for each other layer so that each variable is transformed a similar amount of times.

Let's implement a function that creates a checkerboard mask and a channel mask for us:

```python
[13]:
def create_checkerboard_mask(h, w, invert=False):
    x, y = torch.arange(h, dtype=torch.int32), torch.arange(w, dtype=torch.int32)
    xx, yy = torch.meshgrid(x, y)
    mask = torch.fmod(xx + yy, 2)
    mask = mask.to(torch.float32).view(1, 1, h, w)
    if invert:
        mask = 1 - mask
    return mask

def create_channel_mask(c_in, invert=False):
    mask = torch.cat([torch.ones(c_in//2, dtype=torch.float32),
                      torch.zeros(c_in-c_in//2, dtype=torch.float32)],
                     dim=0)
    mask = mask.view(1, c_in, 1, 1)
    if invert:
        mask = 1 - mask
    return mask
```

We can also visualize the corresponding masks for an image of size $8 \times 8 \times 2$ (2 channels):

```python
[14]:
    checkerboard_mask = create_checkerboard_mask(h=8, w=8).expand(-1, 2, -1, -1)
    channel_mask = create_channel_mask(c_in=2).expand(-1, -1, 8, 8)

    show_imgs(checkerboard_mask.transpose(0, 1), "Checkerboard mask")
    show_imgs(channel_mask.transpose(0, 1), "Channel mask")
```
As a last aspect of coupling layers, we need to decide for the deep neural network we want to apply in the coupling layers. The input to the layers is an image, and hence we stick with a CNN. Because the input to a transformation depends on all transformations before, it is crucial to ensure a good gradient flow through the CNN back to the input, which can be optimally achieved by a ResNet-like architecture. Specifically, we use a Gated ResNet that adds a $\sigma$-gate to the skip connection, similarly to the input gate in LSTMs. The details are not necessarily important here, and the network is strongly inspired from Flow++ [3] in case you are interested in building even stronger models.

```python
[15]: class ConcatELU(nn.Module):
    """
    Activation function that applies ELU in both direction (inverted and plain).
    Allows non-linearity while providing strong gradients for any input (important
    for final convolution)
    """
    def forward(self, x):
        return torch.cat([F.elu(x), F.elu(-x)], dim=1)

class LayerNormChannels(nn.Module):
    def __init__(self, c_in):
        """
        This module applies layer norm across channels in an image. Has been shown to
        work well with ResNet connections.
        Inputs:
        c_in - Number of channels of the input
        """
        super().__init__()
        self.layer_norm = nn.LayerNorm(c_in)
    def forward(self, x):
        x = x.permute(0, 2, 3, 1)
        x = self.layer_norm(x)
        x = x.permute(0, 3, 1, 2)
        return x
```

(continues on next page)
class GatedConv(nn.Module):

    def __init__(self, c_in, c_hidden):
        
        class GatedConvNet(nn.Module):

            def __init__(self, c_in, c_hidden=32, c_out=-1, num_layers=3):
                
                super().__init__()
Training loop

Finally, we can add Dequantization, Variational Dequantization and Coupling Layers together to build our full normalizing flow on MNIST images. We apply 8 coupling layers in the main flow, and 4 for variational dequantization if applied. We apply a checkerboard mask throughout the network as with a single channel (black-white images), we cannot apply channel mask. The overall architecture is visualized below.

```python
[16]: def create_simple_flow(use_vardeq=True):
    flow_layers = []
    if use_vardeq:
        vardeq_layers = [CouplingLayer(network=GatedConvNet(c_in=2, c_out=2, c_
→hidden=16), mask=create_checkerboard_mask(h=28, w=28, c_in=1, invert=(i%2==1))
        for i in range(4)]
        flow_layers += [VariationalDequantization(var_flows=vardeq_layers)]
    else:
        flow_layers += [Dequantization()]
    for i in range(8):
        flow_layers += [CouplingLayer(network=GatedConvNet(c_in=1, c_hidden=32), mask=create_checkerboard_mask(h=28, w=28, c_in=1, invert=(i%2==1))]
    flow_model = ImageFlow(flow_layers).to(device)
    return flow_model
```

For implementing the training loop, we use the framework of PyTorch Lightning and reduce the code overhead. If interested, you can take a look at the generated tensorboard file, in particularly the graph to see an overview of flow transformations that are applied. Note that we again provide pre-trained models (see later on in the notebook) as normalizing flows are particularly expensive to train. We have also run validation and testing as this can take some time as well with the added importance sampling.

```python
[17]: def train_flow(flow, model_name="MNISTFlow"):
    # Create a PyTorch Lightning trainer
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, model_name),
→checkpoint_callback=ModelCheckpoint(save_weights_only=True, mode="min", monitor="val_bpd"),
→gpus=1 if torch.cuda.is_available() else 0,
→max_epochs=200,
→gradient_clip_val=1.0,
→callbacks=[LearningRateMonitor("epoch")]
    trainer.logger._log_graph = True
    trainer.logger._default_hp_metric = None # Optional logging argument that we don’t need
    train_data_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_
→last=True, pin_memory=True, num_workers=8)
    result = None
    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, model_name + ".ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        ckpt = torch.load(pretrained_filename)
```

(continues on next page)
5.13.3 Multi-scale architecture

One disadvantage of normalizing flows is that they operate on the exact same dimensions as the input. If the input is high-dimensional, so is the latent space, which requires larger computational cost to learn suitable transformations. However, particularly in the image domain, many pixels contain less information in the sense that we could remove them without losing the semantic information of the image.

Based on this intuition, deep normalizing flows on images commonly apply a multi-scale architecture [1]. After the first $N$ flow transformations, we split off half of the latent dimensions and directly evaluate them on the prior. The other half is run through $N$ more flow transformations, and depending on the size of the input, we split it again in half or stop overall at this position. The two operations involved in this setup is Squeeze and Split which we will review more closely and implement below.

Squeeze and Split

When we want to remove half of the pixels in an image, we have the problem of deciding which variables to cut, and how to rearrange the image. Thus, the squeezing operation is commonly used before split, which divides the image into subsquares of shape $2 \times 2 \times C$, and reshapes them into $1 \times 1 \times 4C$ blocks. Effectively, we reduce the height and width of the image by a factor of 2 while scaling the number of channels by 4. Afterwards, we can perform the split operation over channels without the need of rearranging the pixels. The smaller scale also makes the overall architecture more efficient. Visually, the squeeze operation should transform the input as follows:

The input of $4 \times 4 \times 1$ is scaled to $2 \times 2 \times 4$ following the idea of grouping the pixels in $2 \times 2 \times 1$ subsquares. Next, let’s try to implement this layer:

```python
[18]: class SqueezeFlow(nn.Module):
    def forward(self, z, ldj, reverse=False):
        B, C, H, W = z.shape
        if not reverse:
            # Forward direction: H x W x C => H/2 x W/2 x 4C
            z = z.reshape(B, C, H//2, 2, W//2, 2)
```
Before moving on, we can verify our implementation by comparing our output with the example figure above:

```python
[19]: sq_flow = SqueezeFlow()
rand_img = torch.arange(1,17).view(1, 1, 4, 4)
print("Image (before)\n", rand_img)
forward_img, _ = sq_flow(rand_img, ldj=None, reverse=False)
print("\nImage (forward)\n", forward_img.permute(0,2,3,1)) # Permute for readability
reconst_img, _ = sq_flow(forward_img, ldj=None, reverse=True)
print("\nImage (reverse)\n", reconst_img)
```

Image (before)
tensor([[ 1, 2, 3, 4],
        [ 5, 6, 7, 8],
        [ 9, 10, 11, 12],
        [13, 14, 15, 16]]))

Image (forward)
tensor([[ 1, 2, 5, 6],
        [ 3, 4, 7, 8]],
        [[ 9, 10, 13, 14],
        [11, 12, 15, 16]])

Image (reverse)
tensor([[ 1, 2, 3, 4],
        [ 5, 6, 7, 8],
        [ 9, 10, 11, 12],
        [13, 14, 15, 16]])

The split operation divides the input into two parts, and evaluates one part directly on the prior. So that our flow operation fits to the implementation of the previous layers, we will return the prior probability of the first part as the log determinant jacobian of the layer. It has the same effect as if we would combine all variable splits at the end of the flow, and evaluate them together on the prior.

```python
[20]: class SplitFlow(nn.Module):
    def __init__(self):
        super().__init__()
        self.prior = torch.distributions.normal.Normal(loc=0.0, scale=1.0)

    def forward(self, z, ldj, reverse=False):
        if not reverse:
            z, z_split = z.chunk(2, dim=1)
            ldj += self.prior.log_prob(z_split).sum(dim=[1,2,3])
        else:
            z_split = self.prior.sample(sample_shape=z.shape).to(device)
            z = torch.cat([z, z_split], dim=1)
            ldj -= self.prior.log_prob(z_split).sum(dim=[1,2,3])
```

(continues on next page)
Building a multi-scale flow

After defining the squeeze and split operation, we are finally able to build our own multi-scale flow. Deep normalizing flows such as Glow and Flow++ [2,3] often apply a split operation directly after squeezing. However, with shallow flows, we need to be more thoughtful about where to place the split operation as we need at least a minimum amount of transformations on each variable. Our setup is inspired by the original RealNVP architecture [1] which is shallower than other, more recent state-of-the-art architectures.

Hence, for the MNIST dataset, we will apply the first squeeze operation after two coupling layers, but don’t apply a split operation yet. Because we have only used two coupling layers and each the variable has been only transformed once, a split operation would be too early. We apply two more coupling layers before finally applying a split flow and squeeze again. The last four coupling layers operate on a scale of $7 \times 7 \times 8$. The full flow architecture is shown below.

Note that while the feature maps inside the coupling layers reduce with the height and width of the input, the increased number of channels is not directly considered. To counteract this, we increase the hidden dimensions for the coupling layers on the squeezed input. The dimensions are often scaled by 2 as this approximately increases the computation cost by 4 canceling with the squeezing operation. However, we will choose the hidden dimensionalities 32, 48, 64 for the three scales respectively to keep the number of parameters reasonable and show the efficiency of multi-scale architectures.

```python
[21]: def create_multiscale_flow():
    flow_layers = []
    vardeq_layers = [CouplingLayer(network=GatedConvNet(c_in=2, c_out=2, c_hidden=16),
        mask=create_checkerboard_mask(h=28, w=28, invert=(i % 2==1)),
        c_in=1) for i in range(4)]
    flow_layers += [VariationalDequantization(vardeq_layers)]
    flow_layers += [CouplingLayer(network=GatedConvNet(c_in=1, c_hidden=32),
        mask=create_checkerboard_mask(h=28, w=28, invert=(i % 2==1)),
        c_in=1) for i in range(2)]
    flow_layers += [SqueezeFlow()]
    for i in range(2):
        flow_layers += [CouplingLayer(network=GatedConvNet(c_in=4, c_hidden=48),
            mask=create_channel_mask(c_in=4, invert=(i % 2==1)),
            c_in=4)]
        flow_layers += [SplitFlow(),
            SqueezeFlow()]
    for i in range(4):
        flow_layers += [CouplingLayer(network=GatedConvNet(c_in=8, c_hidden=64),
            mask=create_channel_mask(c_in=8, invert=(i % 2==1)),
            c_in=8)]
    flow_model = ImageFlow(flow_layers).to(device)
    return flow_model
```

We can show the difference in number of parameters below:
```python
[22]:

def print_num_params(model):
    num_params = sum([np.prod(p.shape) for p in model.parameters()])
    print("Number of parameters: {:,}".format(num_params))

print_num_params(create_simple_flow(use_vardeq=False))
print_num_params(create_simple_flow(use_vardeq=True))
print_num_params(create_multiscale_flow())

Number of parameters: 335,128
Number of parameters: 379,556
Number of parameters: 1,062,090
```

Although the multi-scale flow has almost 3 times the parameters of the single scale flow, it is not necessarily more computationally expensive than its counterpart. We will compare the runtime in the following experiments as well.

### 5.13.4 Analysing the flows

In the last part of the notebook, we will train all the models we have implemented above, and try to analyze the effect of the multi-scale architecture and variational dequantization.

#### Training flow variants

Before we can analyse the flow models, we need to train them first. We provide pre-trained models that contain the validation and test performance, and run-time information. As flow models are computationally expensive, we advice you to rely on those pretrained models for a first run through the notebook.

```python
[23]:

flow_dict = {
    "simple": {},
    "vardeq": {},
    "multiscale": {}
}
flow_dict["simple"]['model'], flow_dict["simple"]['result'] = train_flow(create__simple_flow(use_vardeq=False), model_name="MNISTFlow_simple")
flow_dict["vardeq"]['model'], flow_dict["vardeq"]['result'] = train_flow(create__simple_flow(use_vardeq=True), model_name="MNISTFlow_vardeq")
flow_dict["multiscale"]['model'], flow_dict["multiscale"]['result'] = train__flow(create_multiscale_flow(), model_name="MNISTFlow_multiscale")
```

GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
Found pretrained model, loading...
Found pretrained model, loading...
Found pretrained model, loading...
Density modeling and sampling

Firstly, we can compare the models on their quantitative results. The following table shows all important statistics. The inference time specifies the time needed to determine the probability for a batch of 64 images for each model, and the sampling time the duration it took to sample a batch of 64 images.

As we have initially expected, using variational dequantization improves upon standard dequantization in terms of bits per dimension. Although the difference with 0.04bpd doesn’t seem impressive first, it is a considerably step for generative models (most state-of-the-art models improve upon previous models in a range of 0.02-0.1bpd on CIFAR with three times as high bpd). While it takes longer to evaluate the probability of an image due to the variational dequantization, which also leads to a longer training time, it does not have an effect on the sampling time. This is because inverting variational dequantization is the same as dequantization: finding the next lower integer.

When we compare the two models to multi-scale architecture, we can see that the bits per dimension score again dropped by about 0.04bpd. Additionally, the inference time and sampling time improved notably despite having more parameters. Thus, we see that the multi-scale flow is not only stronger for density modeling, but also more efficient.

Next, we can test the sampling quality of the models. We should note that the samples for variational dequantization and standard dequantization are very similar, and hence we visualize here only the ones for variational dequantization and the multi-scale model. However, feel free to also test out the "simple" model. The seeds are set to obtain reproducible generations and are not cherry picked.
```python
[27]: pl.seed_everything(44)
samples = flow_dict['multiscale']['model'].sample(img_shape=[16,8,7,7])
show_imgs(samples.cpu())
```

```
/home/phillip/anaconda3/envs/nlp1/lib/python3.7/site-packages/ipykernel_launcher.py:
  51: UserWarning: Mixed memory format inputs detected while calling the operator. The operator will output contiguous tensor even if some of the inputs are in channels_last format. (Triggered internally at /pytorch/aten/src/ATen/native/TensorIterator.cpp:918.)
```
From the few samples, we can see a clear difference between the simple and the multi-scale model. The single-scale model has only learned local, small correlations while the multi-scale model was able to learn full, global relations that form digits. This showcases another benefit of the multi-scale model. In contrast to VAEs, the outputs are sharp as normalizing flows can naturally model complex, multi-modal distributions while VAEs have the independent decoder output noise. Nevertheless, the samples from this flow are far from perfect as not all samples show true digits.

Interpolation in latent space

Another popular test for the smoothness of the latent space of generative models is to interpolate between two training examples. As normalizing flows are strictly invertible, we can guarantee that any image is represented in the latent space. We again compare the variational dequantization model with the multi-scale model below.

```
[28]: @torch.no_grad()
def interpolate(model, img1, img2, num_steps=8):
    
    Inputs:
    model - object of ImageFlow class that represents the (trained) flow model
    img1, img2 - Image tensors of shape [1, 28, 28]. Images between which should be interpolated.
    num_steps - Number of interpolation steps. 8 interpolation steps mean 6 intermediate pictures besides img1 and img2
    
    imgs = torch.stack([img1, img2], dim=0).to(model.device)
    z, _ = model.encode(imgs)
    ```
(continues on next page)
alpha = torch.linspace(0, 1, steps=num_steps, device=z.device).view(-1, 1, 1, 1)
interpolations = z[0:1] * alpha + z[1:2] * (1 - alpha)
interp_imgs = model.sample(interpolations.shape[:1] + imgs.shape[1:], z_
   init=interpolations)
show_imgs(interp_imgs, row_size=8)
exmp_imgs, _ = next(iter(train_loader))

[29]: pl.seed_everything(42)
for i in range(2):
    interpolate(flow_dict["vardeq"]["model"], exmp_imgs[2*i], exmp_imgs[2*i+1])

![Images of interpolated digits]

[30]: pl.seed_everything(42)
for i in range(2):
    interpolate(flow_dict["multiscale"]["model"], exmp_imgs[2*i], exmp_imgs[2*i+1])

![Images of interpolated digits]

The interpolations of the multi-scale model result in more realistic digits (first row 7 ↔ 8 ↔ 6, second row 9 ↔ 4 ↔ 6), while the variational dequantization model focuses on local patterns that globally do not form a digit. For the multi-scale model, we actually did not do the “true” interpolation between the two images as we did not consider the variables that were split along the flow (they have been sampled randomly for all samples). However, as we will see in the next experiment, the early variables do not effect the overall image much.
Visualization of latents in different levels of multi-scale

In the following we will focus more on the multi-scale flow. We want to analyse what information is being stored in the variables split at early layers, and what information for the final variables. For this, we sample 8 images where each of them share the same final latent variables, but differ in the other part of the latent variables. Below we visualize three examples of this:

```python
pl.seed_everything(44)
for _ in range(3):
    z_init = flow_dict["multiscale"]["model"].prior.sample(sample_shape=[1,8,7,7])
    z_init = z_init.expand(8, -1, -1, -1)
    samples = flow_dict["multiscale"]["model"].sample(img_shape=z_init.shape, z_init=z_init)
    show_imgs(samples.cpu())
```

5.13. Tutorial 11: Normalizing Flows for image modeling
We see that the early split variables indeed have a smaller effect on the image. Still, small differences can be spotted when we look carefully at the borders of the digits. For instance, the hole at the top of the 8 changes for different samples although all of them represent the same coarse structure. This shows that the flow indeed learns to separate the higher-level information in the final variables, while the early split ones contain local noise patterns.

**Visualizing Dequantization**

As a final part of this notebook, we will look at the effect of variational dequantization. We have motivated variational dequantization by the issue of sharp edges/borders being difficult to model, and a flow would rather prefer smooth, prior-like distributions. To check how what noise distribution $q(u|x)$ the flows in the variational dequantization module have learned, we can plot a histogram of output values from the dequantization and variational dequantization module.

```python
[32]: def visualize_dequant_distribution(model : ImageFlow, imgs : torch.Tensor, title: ...
  ...:   str=None):
  ...:     
  ...:     model - The flow of which we want to visualize the dequantization distribution
  ...:     imgs - Example training images of which we want to visualize the...
  ...:     dequantization distribution
  ...:     
  ...:     imgs = imgs.to(device)
  ...:     ldj = torch.zeros(imgs.shape[0], dtype=torch.float32).to(device)
  ...:     with torch.no_grad():
  ...:         dequant_vals = []
  ...:         for _ in tqdm(range(8), leave=False):
  ...:             d, _ = model.flows[0](imgs, ldj, reverse=False)
  ...:             dequant_vals.append(d)
  ...:             dequant_vals = torch.cat(dequant_vals, dim=0)
  ...:             dequant_vals = dequant_vals.view(-1).cpu().numpy()
  ...:             sns.set()
  ...:             plt.figure(figsize=(10,3))
  ...:             plt.hist(dequant_vals, bins=256, color=to_rgb("C0")+(0.5,)), edgecolor="C0",...
  ...:             if title is not None:
  ...:                 plt.title(title)
  ...:             plt.show()
  ...:             plt.close()
  ...:     sample_imgs, _ = next(iter(train_loader))
```
The dequantization distribution in the first plot shows that the MNIST images have a strong bias towards 0 (black), and the distribution of them have a sharp border as mentioned before. The variational dequantization module has indeed learned a much smoother distribution with a Gaussian-like curve which can be modeled much better. For the other values, we would need to visualize the distribution $q(u|x)$ on a deeper level, depending on $x$. However, as all $u$’s interact and depend on each other, we would need to visualize a distribution in 784 dimensions, which is not that intuitive anymore.
5.13.5 Conclusion

In conclusion, we have seen how to implement our own normalizing flow, and what difficulties arise if we want to apply them on images. Dequantization is a crucial step in mapping the discrete images into continuous space to prevent underisable delta-peak solutions. While dequantization creates hypercubes with hard border, variational dequantization allows us to fit a flow much better on the data. This allows us to obtain a lower bits per dimension score, while not affecting the sampling speed. The most common flow element, the coupling layer, is simple to implement, and yet effective. Furthermore, multi-scale architectures help to capture the global image context while allowing us to efficiently scale up the flow. Normalizing flows are an interesting alternative to VAEs as they allow an exact likelihood estimate in continuous space, and we have the guarantee that every possible input $x$ has a corresponding latent vector $z$. However, even beyond continuous inputs and images, flows can be applied and allow us to exploit the data structure in latent space, as e.g. on graphs for the task of molecule generation [6]. Recent advances in Neural ODEs allow a flow with infinite number of layers, called Continuous Normalizing Flows, whose potential is yet to fully explore. Overall, normalizing flows are an exciting research area which will continue over the next couple of years.

5.13.6 References


5.14 Tutorial 12: Autoregressive Image Modeling

Filled notebook:
Pre-trained models:

In this tutorial, we implement an autoregressive likelihood model for the task of image modeling. Autoregressive models are naturally strong generative models that constitute one of the current state-of-the-art architectures on likelihood-based image modeling, and are also the basis for large language generation models such as GPT3. Similar to the language generation you have seen in assignment 2, autoregressive models work on images by modeling the likelihood of a pixel given all previous ones. For instance, in the picture below, we model the pixel $x_i$ as a conditional probability distribution based on all previous (here blue) pixels (figure credit - Aaron van den Oord et al.):
Generally, autoregressive model over high-dimensional data $x$ factor the joint distribution as the following product of conditionals:

$$p(x) = p(x_1, ..., x_n) = \prod_{i=1}^{n} p(x_i|x_1, ..., x_{i-1})$$

Learning these conditionals is often much simpler than learning the joint distribution $p(x)$ all together. However, disadvantages of autoregressive models include slow sampling, especially for large images, as we need height-times-width forward passes through the model. In addition, for some applications, we require a latent space as modeled in VAEs and Normalizing Flows. For instance, in autoregressive models, we cannot interpolate between two images because of the lack of a latent representation. We will explore and discuss these benefits and drawbacks alongside with our implementation.

Our implementation will focus on the PixelCNN [2] model which has been discussed in detail in the lecture. Most current SOTA models use PixelCNN as their fundamental architecture, and various additions have been proposed to improve the performance (e.g. PixelCNN++ and PixelSNAIL). Hence, implementing PixelCNN is a good starting point for our short tutorial.

First of all, we need to import our standard libraries. Similarly as in the last couple of tutorials, we will use PyTorch Lightning here as well.

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

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

   ## Progress bar
   from tqdm.notebook import tqdm

   ## PyTorch
   import torch
   import torch.nn as nn
   import torch.nn.functional as F
   import torch.utils.data as data
   import torch.optim as optim
   # Torchvision
   import torchvision
   from torchvision.datasets import MNIST
   from torchvision import transforms
   # PyTorch Lightning
   try:
     import pytorch_lightning as pl
   except ModuleNotFoundError:
     # Google Colab does not have PyTorch Lightning installed by default. Hence, we do it here if necessary
     !pip install pytorch-lightning==1.0.3
     import pytorch_lightning as pl
   from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

   # Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial12_dupl"

# Setting the seed
pl.seed_everything(42)

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

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

Using device cuda:0

We again provide a pretrained model, which is downloaded below:

```python
import urllib.request
from urllib.error import HTTPError

# Github URL where saved models are stored for this tutorial

# Files to download
pretrained_files = ['PixelCNN.ckpt']

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

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

```

Similar to the Normalizing Flows in Tutorial 11, we will work on the MNIST dataset and use 8-bits per pixel (values between 0 and 255). The dataset is loaded below:

```python
# Convert images from 0-1 to 0-255 (integers). We use the long datatype as we will use the images as labels as well
def discretize(sample):
    return (sample * 255).to(torch.long)

# Transformations applied on each image => only make them a tensor
transform = transforms.Compose([transforms.ToTensor(),
                                discretize])

# Loading the training dataset. We need to split it into a training and validation part
train_dataset = MNIST(root=DATASET_PATH, train=True, transform=transform, download=True)
```

(continues on next page)
pl.seed_everything(42)
train_set, val_set = torch.utils.data.random_split(train_dataset, [50000, 10000])

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

# We define a set of data loaders that we can use for various purposes later.
train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True, pin_memory=True, num_workers=4)
val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False, drop_last=False, num_workers=4)
test_loader = data.DataLoader(test_set, batch_size=128, shuffle=False, drop_last=False, num_workers=4)

A good practice is to always visualize some data examples to get an intuition of the data:

```py
[4]: def show_imgs(imgs):
    num_imgs = imgs.shape[0] if isinstance(imgs, torch.Tensor) else len(imgs)
    nrow = min(num_imgs, 4)
    ncol = int(math.ceil(num_imgs/nrow))
    imgs = torchvision.utils.make_grid(imgs, nrow=nrow, pad_value=128)
    imgs = imgs.clamp(min=0, max=255)
    np_imgs = imgs.cpu().numpy()
    plt.figure(figsize=(1.5*nrow, 1.5*ncol))
    plt.imshow(np.transpose(np_imgs, (1,2,0)), interpolation='nearest')
    plt.axis('off')
    plt.show()
    plt.close()
show_imgs([train_set[i][0] for i in range(8)])
```

```
6 7 6 9

4 2 1 3
```
5.14.1 Masked autoregressive convolutions

The core module of PixelCNN is its masked convolutions. In contrast to language models, we don’t apply an LSTM on each pixel one-by-one. This would be inefficient because images are grids instead of sequences. Thus, it is better to rely on convolutions that have shown great success in deep CNN classification models.

Nevertheless, we cannot just apply standard convolutions without any changes. Remember that during training of autoregressive models, we want to use teacher forcing which both helps the model training, and significantly reduces the time needed for training. For image modeling, teacher forcing is implemented by using a training image as input to the model, and we want to obtain as output the prediction for each pixel based on only its predecessors. Thus, we need to ensure that the prediction for a specific pixel can only be influenced by its predecessors and not by its own value or any “future” pixels. For this, we apply convolutions with a mask.

Which mask we use depends on the ordering of pixels we decide on, i.e. which is the first pixel we predict, which is the second one, etc. The most commonly used ordering is to denote the upper left pixel as the start pixel, and sort the pixels row by row, as shown in the visualization at the top of the tutorial. Thus, the second pixel is on the right of the first one (first row, second column), and once we reach the end of the row, we start in the second row, first column. If we now want to apply this to our convolutions, we need to ensure that the prediction of pixel 1 is not influenced by its own “true” input, and all pixels on its right and in any lower row. In convolutions, this means that we want to set those entries of the weight matrix to zero that take pixels on the right and below into account. As an example for a 5x5 kernel, see a mask below (figure credit - Aaron van den Oord):

Before looking into the application of masked convolutions in PixelCNN in detail, let’s first implement a module that allows us to apply an arbitrary mask to a convolution:

```python
[5]:
class MaskedConvolution(nn.Module):
    def __init__(self, c_in, c_out, mask, **kwargs):
        """ Implements a convolution with mask applied on its weights. ""
        Inputs:
        c_in - Number of input channels
        c_out - Number of output channels
        mask - Tensor of shape [kernel_size_H, kernel_size_W] with 0s where
        the convolution should be masked, and 1s otherwise.
        **kwargs - Additional arguments for the convolution
        """
        super().__init__()
        # For simplicity: calculate padding automatically
        kernel_size = (mask.shape[0], mask.shape[1])
        dilation = 1 if "dilation" not in kwargs else kwargs["dilation"]
        padding = tuple([dilation*(kernel_size[i]-1)//2 for i in range(2)])
        # Actual convolution
        self.conv = nn.Conv2d(c_in, c_out, kernel_size, padding=padding, **kwargs)
        # Mask as buffer => it is no parameter but still a tensor of the module
        # (must be moved with the devices)
        self.register_buffer('mask', mask
        [None, None])

    def forward(self, x):
        self.conv.weight.data *= self.mask # Ensures zero's at masked positions
        return self.conv(x)
```
Vertical and horizontal convolution stacks

To build our own autoregressive image model, we could simply stack a few masked convolutions on top of each other. This was actually the case for the original PixelCNN model, discussed in the paper *Pixel Recurrent Neural Networks*, but this leads to a considerable issue. When sequentially applying a couple of masked convolutions, the receptive field of a pixel show to have a “blind spot” on the right upper side, as shown in the figure below (figure credit - Aaron van den Oord et al.):

Although a pixel should be able to take into account all other pixels above and left of it, a stack of masked convolutions does not allow us to look to the upper pixels on the right. This is because the features of the pixels above, which we use for convolution, do not contain any information of the pixels on the right of the same row. If they would, we would be “cheating” and actually looking into the future. To overcome this issue, van den Oord et al. [2] proposed to split the convolutions into a vertical and a horizontal stack. The vertical stack looks at all pixels above the current one, while the horizontal takes into account all on the left. While keeping both of them separate, we can actually look at the pixels on the right with the vertical stack without breaking any of our assumptions. The two convolutions are also shown in the figure above.

Let us implement them here as follows:

```python
[6]: class VerticalStackConvolution(MaskedConvolution):
    def __init__(self, c_in, c_out, kernel_size=3, mask_center=False, **kwargs):
        # Mask out all pixels below. For efficiency, we could also reduce the kernel
        # size in height, but for simplicity, we stick with masking here.
        mask = torch.ones(kernel_size, kernel_size)
        mask[kernel_size//2+1:, :] = 0

        # For the very first convolution, we will also mask the center row
        if mask_center:
            mask[kernel_size//2, :] = 0

        super().__init__(c_in, c_out, mask, **kwargs)

class HorizontalStackConvolution(MaskedConvolution):
    def __init__(self, c_in, c_out, kernel_size=3, mask_center=False, **kwargs):
        # Mask out all pixels on the left. Note that our kernel has a size of 1
        # in height because we only look at the pixel in the same row.
        mask = torch.ones(1,kernel_size)
        mask[0,kernel_size//2+1:] = 0

        # For the very first convolution, we will also mask the center pixel
        if mask_center:
            mask[0,kernel_size//2] = 0

        super().__init__(c_in, c_out, mask, **kwargs)
```

Note that we have an input argument called `mask_center`. Remember that the input to the model is the actual input image. Hence, the very first convolution we apply cannot use the center pixel as input, but must be masked. All consecutive convolutions, however, should use the center pixel as we otherwise lose the features of the previous layer. Hence, the input argument `mask_center` is True for the very first convolutions, and False for all others.
Visualizing the receptive field

To validate our implementation of masked convolutions, we can visualize the receptive field we obtain with such convolutions. We should see that with increasing number of convolutional layers, the receptive field grows in both vertical and horizontal direction, without the issue of a blind spot. The receptive field can be empirically measured by backpropagating an arbitrary loss for the output features of a specific pixel with respect to the input. We implement this idea below, and visualize the receptive field below.

```python
[7]: inp_img = torch.zeros(1, 1, 11, 11)
inp_img.requires_grad_()

def show_center_recep_field(img, out):
    
    """
    Calculates the gradients of the input with respect to the output center pixel, 
    and visualizes the overall receptive field.
    Inputs:
    img - Input image for which we want to calculate the receptive field on.
    out - Output features/loss which is used for backpropagation, and should be 
    the output of the network/computation graph.
    """
    # Determine gradients
    loss = out[0,:,img.shape[2]//2,img.shape[3]//2].sum() # L1 loss for simplicity
    loss.backward,retain_graph=True) # Retain graph as we want to stack multiple
    # layers and show the receptive field of all of them
    img_grads = img.grad.abs()
    img.grad.fill_(0) # Reset grads
    
    # Plot receptive field
    img = img_grads.squeeze().cpu().numpy()
    fig, ax = plt.subplots(1,2)
    pos = ax[0].imshow(img)
    ax[1].imshow(img>0)
    # Mark the center pixel in red if it doesn't have any gradients (should be the
    # case for standard autoregressive models)
    show_center = (img[img.shape[0]//2, img.shape[1]//2] == 0)
    if show_center:
        center_pixel = np.zeros(img.shape + (4,))
        center_pixel[center_pixel.shape[0]//2, center_pixel.shape[1]//2,:,:] = np.
        array([1.0, 0.0, 0.0, 1.0])
        for i in range(2):
            ax[i].axis('off')
            if show_center:
                ax[i].imshow(center_pixel)
    ax[0].set_title("Weighted receptive field")
    ax[1].set_title("Binary receptive field")
    plt.show()
    plt.close()

show_center_recep_field(inp_img, inp_img)
```
Let’s first visualize the receptive field of a horizontal convolution without the center pixel. We use a small, arbitrary input image (11 × 11 pixels), and calculate the loss for the center pixel. For simplicity, we initialize all weights with 1 and the bias with 0, and use a single channel. This is sufficient for our visualization purposes.

```python
[8]: horiz_conv = HorizontalStackConvolution(c_in=1, c_out=1, kernel_size=3, mask_center=True)
   horiz_conv.conv.weight.data.fill_(1)
   horiz_conv.conv.bias.data.fill_(0)
   horiz_img = horiz_conv(inp_img)
   show_center_recep_field(inp_img, horiz_img)
```

The receptive field is shown in yellow, the center pixel in red, and all other pixels outside of the receptive field are dark blue. As expected, the receptive field of a single horizontal convolution with the center pixel masked and a 3 × 3 kernel is only the pixel on the left. If we use a larger kernel size, more pixels would be taken into account on the left.

Next, let’s take a look at the vertical convolution:

```python
[9]: vert_conv = VerticalStackConvolution(c_in=1, c_out=1, kernel_size=3, mask_center=True)
   vert_conv.conv.weight.data.fill_(1)
   vert_conv.conv.bias.data.fill_(0)
   vert_img = vert_conv(inp_img)
   show_center_recep_field(inp_img, vert_img)
```
The vertical convolution takes all pixels above into account. Combining these two, we get the L-shaped receptive field of the original masked convolution:

\[ \text{horiz} = \text{vert} + \text{horiz} \]

if we stack multiple horizontal and vertical convolutions, we need to take two aspects into account:

1. The center should not be masked anymore for the following convolutions as the features at the pixel’s position are already independent of its actual value. If it is hard to imagine why we can do this, just change the value below to `mask_center=True` and see what happens.

2. The vertical convolution is not allowed to work on features from the horizontal convolution. In the feature map of the horizontal convolutions, a pixel contains information about all of the “true” pixels on the left. If we apply a vertical convolution which also uses features from the right, we effectively expand our receptive field to the true input which we want to prevent. Thus, the feature maps can only be merged for the horizontal convolution.

Using this, we can stack the convolutions in the following way. We have two feature streams: one for the vertical stack, and one for the horizontal stack. The horizontal convolutions can operate on the joint features of the previous horizontals and vertical convolutions, while the vertical stack only takes its own previous features as input. For a quick implementation, we can therefore sum the horizontal and vertical output features at each layer, and use those as final output features to calculate the loss on. An implementation of 4 consecutive layers is shown below. Note that we reuse the features from the other convolutions with `mask_center=True` from above.
# Initialize convolutions with equal weight to all input pixels

```python
horiz_conv = HorizontalStackConvolution(c_in=1, c_out=1, kernel_size=3, mask_center=False)
horiz_conv.conv.weight.data.fill_(1)
horiz_conv.conv.bias.data.fill_(0)
vert_conv = VerticalStackConvolution(c_in=1, c_out=1, kernel_size=3, mask_center=False)
vert_conv.conv.weight.data.fill_(1)
vert_conv.conv.bias.data.fill_(0)
```

# We reuse our convolutions for the 4 layers here. Note that in a standard network, we don't do that, and instead learn 4 separate convolution. As this cell is only for visualization purposes, we reuse the convolutions for all layers.

```python
for l_idx in range(4):
    vert_img = vert_conv(vert_img)
    horiz_img = horiz_conv(horiz_img) + vert_img
    print("Layer \%i" % (l_idx+2))
    show_center_recep_field(inp_img, horiz_img)
```

Layer 2

<table>
<thead>
<tr>
<th>Weighted receptive field</th>
<th>Binary receptive field</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Weighted receptive field" /></td>
<td><img src="image2.png" alt="Binary receptive field" /></td>
</tr>
</tbody>
</table>

Layer 3

<table>
<thead>
<tr>
<th>Weighted receptive field</th>
<th>Binary receptive field</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Weighted receptive field" /></td>
<td><img src="image4.png" alt="Binary receptive field" /></td>
</tr>
</tbody>
</table>

Layer 4

<table>
<thead>
<tr>
<th>Weighted receptive field</th>
<th>Binary receptive field</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image5.png" alt="Weighted receptive field" /></td>
<td><img src="image6.png" alt="Binary receptive field" /></td>
</tr>
</tbody>
</table>
The receptive field above it visualized for the horizontal stack, which includes the features of the vertical convolutions. It grows over layers without any blind spot as we had before. The difference between “weighted” and “binary” receptive field is that for the latter, we check whether there are any gradients flowing back to this pixel. This indicates that the center pixel indeed can use information from this pixel. Nevertheless, due to the convolution weights, some pixels have a stronger effect on the prediction than others. This is visualized in the weighted receptive field by plotting the gradient magnitude for each pixel instead of a binary yes/no.

Another receptive field we can check is the one for the vertical stack as the one above is for the horizontal stack. Let’s visualize it below:

```python
[12]: show_center_recep_field(inp_img, vert_img)
```
As we have discussed before, the vertical stack only looks at pixels above the one we want to predict. Hence, we can validate that our implementation works as we initially expected it to. As a final step, let’s clean up the computation graph we still had kept in memory for the visualization of the receptive field:

```
[13]: del inp_img, horiz_conv, vert_conv
```

### 5.14.2 Gated PixelCNN

In the next step, we will use the masked convolutions to build a full autoregressive model, called Gated PixelCNN. The difference between the original PixelCNN and Gated PixelCNN is the use of separate horizontal and vertical stacks. However, in literature, you often see that people refer to the Gated PixelCNN simply as “PixelCNN”. Hence, in the following, if we say “PixelCNN”, we usually mean the gated version. What “Gated” refers to in the model name is explained next.

#### Gated Convolutions

For visualizing the receptive field, we assumed a very simplified stack of vertical and horizontal convolutions. Obviously, there are more sophisticated ways of doing it, and PixelCNN uses gated convolutions for this. Specifically, the Gated Convolution block in PixelCNN looks as follows (figure credit - Aaron van den Oord et al.):

The left path is the vertical stack (the $N \times N$ convolution is masked correspondingly), and the right path is the horizontal stack. Gated convolutions are implemented by having a twice as large output channel size, and combine them by a element-wise multiplication of tanh and a sigmoid. For a linear layer, we can express a gated activation unit as follows:

$$ y = \tanh(W_f x) \odot \sigma(W_g x) $$

For simplicity, biases have been neglected and the linear layer split into two part, $W_f$ and $W_g$. This concept resembles the input and modulation gate in an LSTM, and has been used in many other architectures as well. The main motivation behind this gated activation is that it might allow to model more complex interactions and simplifies learning. But as in any other architecture, this is mostly a design choice and can be considered a hyperparameters.

Besides the gated convolutions, we also see that the horizontal stack uses a residual connection while the vertical stack does not. This is because we use the output of the horizontal stack for prediction. Each convolution in the vertical
stack also receives a strong gradient signal as it is only two $1 \times 1$ convolutions away from the residual connection, and does not require another residual connection to all its earlier layers.

The implementation in PyTorch is fairly straightforward for this block, because the visualization above gives us a computation graph to follow:

```python
[14]: class GatedMaskedConv(nn.Module):
    def __init__(self, c_in, **kwargs):
        ""
        Gated Convolution block implemented the computation graph shown above.
        ""
        super().__init__()
        self.conv_vert = VerticalStackConvolution(c_in, c_out=2*c_in, **kwargs)
        self.conv_horiz = HorizontalStackConvolution(c_in, c_out=2*c_in, **kwargs)
        self.conv_vert_to_horiz = nn.Conv2d(2*c_in, 2*c_in, kernel_size=1, padding=0)
        self.conv_horiz_1x1 = nn.Conv2d(c_in, c_in, kernel_size=1, padding=0)
    def forward(self, v_stack, h_stack):
        # Vertical stack (left)
        v_stack_feat = self.conv_vert(v_stack)
        v_val, v_gate = v_stack_feat.chunk(2, dim=1)
        v_stack_out = torch.tanh(v_val) * torch.sigmoid(v_gate)
        # Horizontal stack (right)
        h_stack_feat = self.conv_horiz(h_stack)
        h_stack_feat = h_stack_feat + self.conv_vert_to_horiz(v_stack_feat)
        h_val, h_gate = h_stack_feat.chunk(2, dim=1)
        h_stack_feat = torch.tanh(h_val) * torch.sigmoid(h_gate)
        h_stack_out = self.conv_horiz_1x1(h_stack_feat)
        h_stack_out = h_stack_out + h_stack
        return v_stack_out, h_stack_out
```

Building the model

Using the gated convolutions, we can now build our PixelCNN model. The architecture consists of multiple stacked GatedMaskedConv blocks, where we add an additional dilation factor to a few convolutions. This is used to increase the receptive field of the model and allows to take a larger context into account during generation. As a reminder, dilation on a convolution works looks as follows (figure credit - Vincent Dumoulin and Francesco Visin):

Note that the smaller output size is only because the animation assumes no padding. In our implementation, we will pad the input image correspondingly. Alternatively to dilated convolutions, we could downsample the input and use a encoder-decoder architecture as in PixelCNN++ [3]. This is especially beneficial if we want to build a very deep autoregressive model. Nonetheless, as we seek to train a reasonably small model, dilated convolutions are the more efficient option to use here.

Below, we implement the PixelCNN model as a PyTorch Lightning module. Besides the stack of gated convolutions, we also have the initial horizontal and vertical convolutions which mask the center pixel, and a final $1 \times 1$ convolution which maps the output features to class predictions. To determine the likelihood of a batch of images, we first create our initial features using the masked horizontal and vertical input convolution. Next, we forward the features through the stack of gated convolutions. Finally, we take the output features of the horizontal stack, and apply the $1 \times 1$ convolution for classification. We use the bits per dimension metric for the likelihood, similarly to Tutorial 11 and assignment 3.
```python
[15]: class PixelCNN(pl.LightningModule):
    def __init__(self, c_in, c_hidden):
        super().__init__()
        self.save_hyperparameters()
        # Initial convolutions skipping the center pixel
        self.conv_vstack = VerticalStackConvolution(c_in, c_hidden, mask_center=True)
        self.conv_hstack = HorizontalStackConvolution(c_in, c_hidden, mask_center=True)
        # Convolution block of PixelCNN. We use dilation instead of downscaling
        self.conv_layers = nn.ModuleList([
            GatedMaskedConv(c_hidden),
            GatedMaskedConv(c_hidden, dilation=2),
            GatedMaskedConv(c_hidden),
            GatedMaskedConv(c_hidden, dilation=4),
            GatedMaskedConv(c_hidden),
            GatedMaskedConv(c_hidden, dilation=2),
            GatedMaskedConv(c_hidden)
        ])  
        # Output classification convolution (1x1)
        self.conv_out = nn.Conv2d(c_hidden, c_in * 256, kernel_size=1, padding=0)
        self.example_input_array = train_set[0][0][None]

    def forward(self, x):
        """Forward image through model and return logits for each pixel.
        Inputs:
        x - Image tensor with integer values between 0 and 255.
        """
        # Scale input from 0 to 255 back to -1 to 1
        x = (x.float() / 255.0) * 2 - 1
        # Initial convolutions
        v_stack = self.conv_vstack(x)
        h_stack = self.conv_hstack(x)
        # Gated Convolutions
        for layer in self.conv_layers:
            v_stack, h_stack = layer(v_stack, h_stack)
        # 1x1 classification convolution
        out = self.conv_out(F.elu(h_stack))
        # Output dimensions: [Batch, Classes, Channels, Height, Width]
        out = out.reshape(out.shape[0], 256, out.shape[1]//256, out.shape[2], out.shape[3])
        return out

    def calc_likelihood(self, x):
        # Forward pass with bpd likelihood calculation
        pred = self.forward(x)
        nll = F.cross_entropy(pred, x, reduction='none')
        bpd = nll.mean(dim=[1,2,3]) * np.log2(np.exp(1))
        return bpd.mean()

@torch.no_grad()
```

To sample from the autoregressive model, we need to iterate over all dimensions of the input. We start with an empty image, and fill the pixels one by one, starting from the upper left corner. Note that as for predicting $x_i$, all pixels below it have no influence on the prediction. Hence, we can cut the image in height without changing the prediction while increasing efficiency. Nevertheless, all the loops in the sampling function already show that it will take us quite some time to sample. A lot of computation could be reused across loop iterations as those the features on the already predicted pixels will not change over iterations. Nevertheless, this takes quite some effort to implement, and is often not done in implementations because in the end, autoregressive sampling remains sequential and slow. Hence, we settle with the default implementation here.

Before training the model, we can check the full receptive field of the model on an MNIST image of size $28 \times 28$:
Weighted receptive field

Binary receptive field

The visualization shows that for predicting any pixel, we can take almost half of the image into account. However, keep in mind that this is the “theoretical” receptive field and not necessarily the effective receptive field, which is usually much smaller. For a stronger model, we should therefore try to increase the receptive field even further. Especially, for the pixel on the bottom right, the very last pixel, we would be allowed to take into account the whole image. However, our current receptive field only spans across 1/4 of the image. An encoder-decoder architecture can help with this, but it also shows that we require a much deeper, more complex network in autoregressive models than in VAEs or energy-based models.

Training loop

To train the model, we again can rely on PyTorch Lightning and write a function below for loading the pretrained model if it exists. To reduce the computational cost, we have saved the validation and test score in the checkpoint already:

```python
def train_model(**kwargs):
    # Create a PyTorch Lightning trainer with the generation callback
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "PixelCNN"),
                         checkpoint_callback=ModelCheckpoint(save_weights_only=True,
                         monitor="val_bpd"),
                         gpus=1,
                         max_epochs=150,
                         callbacks=[LearningRateMonitor("epoch")])

    result = None
    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "PixelCNN.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = PixelCNN.load_from_checkpoint(pretrained_filename)
        ckpt = torch.load(pretrained_filename)
        result = ckpt.get("result", None)
    else:
        # (continues on next page)
```
Training the model is time consuming and we recommend using the provided pre-trained model for going through this notebook. However, feel free to play around with the hyperparameter like number of layers etc. if you want to get a feeling for those.

When calling the training function with a pre-trained model, we automatically load it and print its test performance:

```python
[18]: model, result = train_model(c_in=1, c_hidden=64)
```

```
print("Test bits per dimension: %.3f bpd" % (result["test_loss"] if "test_loss" in result else result["test_bpd"]))
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Found pretrained model, loading...
Test bits per dimension: 0.808 bpd
```

With a test performance of 0.809bpd, the PixelCNN significantly outperforms the normalizing flows we have seen in Tutorial 11. Considering image modeling as an autoregressive problem simplifies the learning process as predicting one pixel given the ground truth of all others is much easier than predicting all pixels at once. In addition, PixelCNN can explicitly predict the pixel values by a discrete softmax while Normalizing Flows have to learn transformations in continuous latent space. These two aspects allow the PixelCNN to achieve a notably better performance.

To fully compare the models, let’s also measure the number of parameters of the PixelCNN:

```python
[19]: num_params = sum([np.prod(param.shape) for param in model.parameters()])
print("Number of parameters: {:,}").format(num_params))
```

```
Number of parameters: 852,160
```

Compared to the multi-scale normalizing flows, the PixelCNN has considerably less parameters. Of course, the number of parameters depend on our hyperparameter choices. Nevertheless, in general, it can be said that autoregressive models require considerably less parameters than normalizing flows to reach good performance, based on the reasons stated above. Still, autoregressive models are much slower in sampling than normalizing flows, which limits their possible applications.
5.14.3 Sampling

One way of qualitatively analysing generative models is by looking at the actual samples. Let’s therefore use our sampling function to generate a few digits:

```python
pl.seed_everything(1)
samples = model.sample(img_shape=(16,1,28,28))
show_imgs(samples.cpu())
```

Most of the samples can be identified as digits, and overall we achieve a better quality than we had in normalizing flows. This goes along with the lower likelihood we achieved with autoregressive models. Nevertheless, we also see that there is still place for improvement as a considerable amount of samples cannot be identified (for example the first row). Deeper autoregressive models are expected to achieve better quality, as they can take more context into account for generating the pixels.

Note that on Google Colab, you might see different results, specifically with a white line at the top. After some debugging, it seemed that the difference occurs inside the dilated convolution, as it gives different results for different batch sizes. However, it is hard to debug this further as it might be a bug of the installed PyTorch version on Google Colab.

The trained model itself is not restricted to any specific image size. However, what happens if we actually sample a larger image than we had seen in our training dataset? Let’s try below to sample images of size $64 \times 64$ instead of $28 \times 28$:
The larger images show that changing the size of the image during testing confuses the model and generates abstract figures (you can sometimes spot a digit in the upper left corner). In addition, sampling for images of 64x64 pixels take more than a minute on a GPU. Clearly, autoregressive models cannot be scaled to large images without changing the sampling procedure such as with forecasting. Our implementation is also not the most efficient as many computations can be stored and reused throughout the sampling process. Nevertheless, the sampling procedure stays sequential which is inherently slower than parallel generation like done in normalizing flows.

### Autocompletion

One common application done with autoregressive models is auto-completing an image. As autoregressive models predict pixels one by one, we can set the first $N$ pixels to predefined values and check how the model completes the image. For implementing this, we just need to skip the iterations in the sampling loop that already have a value unequal -1. See above in our PyTorch Lightning module for the specific implementation. In the cell below, we randomly take three images from the training set, mask about the lower half of the image, and let the model autocomplete it. To see the diversity of samples, we do this 12 times for each image:

```python
[22]: def autocomplete_image(img):
    # Remove lower half of the image
    img_init = img.clone()
    img_init[:,10:,:] = -1
    print("Original image and input image to sampling:")
    show_imgs([img, img_init])
    # Generate 12 example completions
    img_init = img_init.unsqueeze(dim=0).expand(12,-1,-1,-1).to(device)
    pl.seed_everything(1)
    img_generated = model.sample(img_init.shape, img_init)
    print("Autocompletion samples:")
    show_imgs(img_generated)

    for i in range(1,4):
        img = train_set[i][0]
        autocomplete_image(img)
```

**Original image and input image to sampling:**

<table>
<thead>
<tr>
<th><img src="#" alt="Image 1" /></th>
<th><img src="#" alt="Image 2" /></th>
</tr>
</thead>
</table>

**Autocompletion samples:**

<table>
<thead>
<tr>
<th><img src="#" alt="Sample 1" /></th>
<th><img src="#" alt="Sample 2" /></th>
<th><img src="#" alt="Sample 3" /></th>
<th><img src="#" alt="Sample 4" /></th>
<th><img src="#" alt="Sample 5" /></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><img src="#" alt="Sample 6" /></th>
<th><img src="#" alt="Sample 7" /></th>
<th><img src="#" alt="Sample 8" /></th>
<th><img src="#" alt="Sample 9" /></th>
<th><img src="#" alt="Sample 10" /></th>
</tr>
</thead>
</table>

**Original image and input image to sampling:**

<table>
<thead>
<tr>
<th><img src="#" alt="Image 3" /></th>
<th><img src="#" alt="Image 4" /></th>
</tr>
</thead>
</table>

**Autocompletion samples:**

<table>
<thead>
<tr>
<th><img src="#" alt="Sample 11" /></th>
<th><img src="#" alt="Sample 12" /></th>
<th><img src="#" alt="Sample 13" /></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><img src="#" alt="Sample 14" /></th>
<th><img src="#" alt="Sample 15" /></th>
<th><img src="#" alt="Sample 16" /></th>
<th><img src="#" alt="Sample 17" /></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><img src="#" alt="Sample 18" /></th>
<th><img src="#" alt="Sample 19" /></th>
<th><img src="#" alt="Sample 20" /></th>
</tr>
</thead>
</table>
Original image and input image to sampling:

HBox(children=(FloatProgress(value=0.0, max=28.0), HTML(value='')))

Autocompletion samples:
For the first two digits (7 and 6), we see that the 12 samples all result in a shape which resemble the original digit. Nevertheless, there are some style difference in writing the 7, and some deformed sixes in the samples. When auto-completing the 9 below, we see that the model can fit multiple digits to it. We obtain diverse samples from 0, 3, 8 and 9. This shows that despite having no latent space, we can still obtain diverse samples from an autoregressive model.

**Visualization of the predictive distribution (softmax)**

Autoregressive models use a softmax over 256 values to predict the next pixel. This gives the model a large flexibility as the probabilities for each pixel value can be learned independently if necessary. However, the values are actually not independent because the values 32 and 33 are much closer than 32 and 255. In the following, we visualize the softmax distribution that the model predicts to gain insights how it has learned the relationships of close-by pixels.

To do this, we first run the model on a batch of images and store the output softmax distributions:

```
[23]: det_loader = data.DataLoader(train_set, batch_size=128, shuffle=False, drop_last=False)
imgs, _ = next(iter(det_loader))
imgs = imgs.to(device)
with torch.no_grad():
    out = model(imgs)
    out = F.softmax(out, dim=1)
mean_out = out.mean(dim=[0,2,3,4]).cpu().numpy()
out = out.cpu().numpy()
```

Before diving into the model, let’s visualize the distribution of the pixel values in the whole dataset:

```
[24]: sns.set()
plot_args = {"color": to_rgb("C0")+(0.5,), "edgecolor": "C0", "linewidth": 0.5, "width": 1.0}
plt.hist(imgs.view(-1).cpu().numpy(), bins=256, density=True, **plot_args)
plt.yscale("log")
```
(continues on next page)
As we would expect from the seen images, the pixel value 0 (black) is the dominant value, followed by a batch of values between 250 and 255. Note that we use a log scale on the y-axis due to the big imbalance in the dataset. Interestingly, the pixel values 64, 128 and 191 also stand out which is likely due to the quantization used during the creation of the dataset. For RGB images, we would also see two peaks around 0 and 255, but the values in between would be much more frequent than in MNIST (see Figure 1 in the PixelCNN++ for a visualization on CIFAR10).

Next, we can visualize the distribution our model predicts (in average):

```python
[25]: plt.bar(np.arange(mean_out.shape[0]), mean_out, **plot_args)
plt.yscale("log")
plt.xticks([0,64,128,192,256])
plt.show()
plt.close()
```
This distribution is very close to the actual dataset distribution. This is in general a good sign, but we can see a slightly smoother histogram than above.

Finally, to take a closer look at learned value relations, we can visualize the distribution for individual pixel predictions to get a better intuition. For this, we pick 4 random images and pixels, and visualize their distribution below:

```python
fig, ax = plt.subplots(2,2, figsize=(10,6))
for i in range(4):
    ax_sub = ax[i//2][i%2]
    ax_sub.bar(np.arange(out.shape[1], dtype=np.int32), out[i+4,:,0,14,14], **plot_args)
    ax_sub.set_yscale("log")
    ax_sub.set_xticks([0,64,128,192,256])
plt.show()
plt.close()
```
Overall we see a very diverse set of distributions, with a usual peak for 0 and close to 1. However, the distributions in the first row show a potentially undesirable behavior. For instance, the value 242 has a 1000x lower likelihood than 243 although they are extremely close and can often not be distinguished. This shows that the model might have not generalized well over pixel values. The better solution to this problem is to use discrete logistics mixtures instead of a softmax distribution. A discrete logistic distribution can be imagined as discretized, binned Gaussians. Using a mixture of discrete logistics instead of a softmax introduces an inductive bias to the model to assign close-by values similar likelihoods. We can visualize a discrete logistic below:

```python
mu = torch.Tensor([128])
sigma = torch.Tensor([2.0])

def discrete_logistic(x, mu, sigma):
    return torch.sigmoid((x+0.5-mu)/sigma) - torch.sigmoid((x-0.5-mu)/sigma)

x = torch.arange(256)
p = discrete_logistic(x, mu, sigma)

# Visualization
plt.figure(figsize=(6,3))
plt.bar(x.numpy(), p.numpy(), **plot_args)
plt.xlim(96,160)
plt.title("Discrete logistic distribution")
plt.xlabel("Pixel value")
plt.ylabel("Probability")
plt.show()
plt.close()
```
Instead of the softmax, the model would output mean and standard deviations for the \( K \) logistics we use in the mixture. This is one of the improvements in autoregressive models that PixelCNN++ [3] has introduced compared to the original PixelCNN.

### 5.14.4 Conclusion

In this tutorial, we have looked at autoregressive image modeling, and implemented the PixelCNN architecture. With the usage of masked convolutions, we are able to apply a convolutional network in which a pixel is only influenced by all its predecessors. Separating the masked convolution into a horizontal and vertical stack allowed us to remove the known blind spot on the right upper row of a pixel. In experiments, autoregressive models outperformed normalizing flows in terms of bits per dimension, but are much slower to sample from. Improvements, that we have not implemented ourselves here, are discrete logistic mixtures, a downsampling architecture, and changing the pixel order in a diagonal fashion (see PixelSNAIL). Overall, autoregressive models are another, strong family of generative models, which however are mostly used in sequence tasks because of their linear scaling in sampling time than quadratic as on images.

### 5.14.5 References

