ASCI CBL Practicals

Release v1.0

Phillip Lippe

May 11, 2022

PRACTICALS

Course website: http://computervisionbylearning.info/
Course edition: May, 2022
Repository: https://github.com/phlippe/asci_cbl_practicals/
Author: Phillip Lippe

UPDATE (May 11): To reduce the workload, we decided that each group only has to do Practical 1-3, and choose between Practical 4 **or** Practical 5, but don't require to do both. Each group can individually choose one of the two advanced practicals based on their interests, and are allowed to skip the other practical.

This website provides you access to the content that we will use for the practical session at the ASCI course "Computer Vision by Learning". In the course, you will conduct 5 practicals to different topics in the domain of machine and deep learning for Computer Vision, with increasing complexity to recent research topics in the field. Throughout the course, you will be guided by Teaching Assistant that help you with any question you may have. The teaching team consists of: Adeel Pervez, David Knigge, David Zhang, Tao Hu, Yunhua Zhang, Zenglin Shi, and Phillip Lippe.

The practicals require familiarity with the Deep Learning framework "PyTorch" and common scientific computing packages of Python such as numpy. If you are not familiar with PyTorch or would like to have a refresher, please check out our tutorial *Introduction to PyTorch* before the course. Additionally, the practicals will require you to train several deep learning models. While we tried to reduce the computational cost of each training as much as possible, you will require access to a GPU in order to finish the practicals. For more details, please check the instruction in the section *How to run the notebooks*, and make sure to have your preferred solution setup and ready to go at the start of the course.

The practicals are intended to be solved in **groups of 2 students**. You can already find team mates before the course starts if you know fellow students, or form the groups during the first practicals. Once you have formed groups, please sign up your group in this Google spreadsheet.

For any remaining questions regarding the practicals, please contact us at p.lippe@uva.nl.

CHAPTER ONE

SCHEDULE

Date	Practical
Monday, 9. May 2022 Tuesday, 10. May 2022	Practical 1: Multi-Layer Perceptrons Practical 2: Convolutional Neural Networks Practical 3: Vision Transformers
Wednesday, 11. May 2022	Practical 4: Regular Group Convolutions
Thursday, 12. May 2022	Practical 5: Self-Supervised Contrastive Learning

The 5 practicals are aligned with the lectures of each day. For the first two days, it is expected that you finish the first three practicals. Ideally, you would finish Practical 1 and start with Practical 2 on Monday, and finish Practical 2 as well as Practical 3 on Tuesday. For the remaining two days, one practical per day is scheduled which aligns with the first lecture of each day. On each day, we have a practical session in the Hotel Casa from 13.30-17.00, where TAs will be in the room to answer your questions.

CHAPTER

TWO

HOW TO RUN THE NOTEBOOKS

On this website, you will find the notebooks exported into a HTML format so that you can read them from whatever device you prefer. Your task is to fill in the notebooks to solve the practicals. There are three main ways of running the notebooks we recommend:

- **Locally on GPU**: If you have a laptop with a build-in NVIDIA GPU, we recommend that you run the practicals on your own machine. All notebooks are stored on the github repository that also builds this website. You can find them here: https://github.com/phlippe/asci_cbl_practicals/. While Practical 1 can be executed on common laptops without a GPU, the later practicals require access to a GPU to keep the training times in a reasonable range. Nonetheless, if you prefer, you can code and test most of your code on a CPU-only system, i.e. your own laptop, and once your code is tested and ready, use one of the remaining options to train the model. To ensure that you have all the right python packages installed, we provide a conda environment in the same repository.
- **Google Colab**: If you do not have access to a GPU on your local machine, you can make use of Google Colab. Google Colab provides you access to GPUs for free, and you can activate the GPU support by `Runtime -> Change runtime type -> Hardware accelerator: GPU`. Each notebook on this documentation website has a badge with a link to directly open it on Google Colab. It is highly recommend to copy the notebook to your own Google Drive before starting, since 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. In addition, note that for free account, Google Colab is limited to one session at a time, and each session has a time limit.
- **Compute cluster**: If you have access to a compute cluster, we recommend to using it to do your final trainings. Depending on your preference, you can implement the practicals either locally or on Google Colab. Once your notebook is ready, you can first convert the notebooks to a script using `jupyter nbconvert --to script ...ipynb`, and then start a job on the cluster for running the script. A few advices when running on clusters:
  - Disable the tqdm statements in the notebook. Otherwise your slurm output file might overflow and be several MB large.
  - Comment out the matplotlib plotting statements, or change `plt.show()` to `plt.savefig(...)`.

CHAPTER

THREE

REPORT SUBMISSION

Submission email: asci.cbl.practicals@googlemail.com
Deadline: 31. May, 2022 (23:59 CEST)

At the end of the course, you are expected to submit a report about your findings of your practicals. The report should be prepared as a PDF in the LaTeX template provided in the repository at *report/*. In the report, you are expected to answer the questions in the practicals, and include any figures requested in the practicals (e.g. the training and/or evaluation curves of a model). Your report should have roughly 1 page per practical, with a maximum of 8 pages. For submission, add your report to your filled notebooks as a zip file and send it to asci.cbl.practicals@googlemail.com. Please make sure to **clear all outputs of the notebooks** before submitting, and **exclude any datasets or trained models from your submission**.

Please remember that we do not allow any form of plagiarism. Any plagiarism will lead to all team members failing the course.

3.1 Tutorial: Introduction to PyTorch

Open notebook:
Recordings:

Welcome to our PyTorch tutorial for the ASCI course "Computer Vision by Learning" 2022! 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 environment and have activated it.

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

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

## Progress bar
from tqdm.notebook import tqdm
```

3.1.1 The Basics of PyTorch

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

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

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

```
Using torch 1.10.0
```

At the time of writing this tutorial (mid of April 2022), the current stable version is 1.11. You should therefore see the output `Using torch 1.11.0` or `Using torch 1.10.0`, eventually with some extension for the CUDA version on Colab. In general, it is recommended to keep the PyTorch version updated to the newest one. If you see a lower version number than 1.10, make sure you have installed the correct the environment, or ask one of your TAs. In case PyTorch 1.12 or newer will be published during the time of the course, don't worry. The interface between PyTorch versions doesn't change too much, and hence all code should also be runnable with newer versions.

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

[3]:
```
torch.manual_seed(42) # Setting the seed
```

[3]:
```
<torch._C.Generator at 0x7fad896113d0>
```

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:

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

```
tensor([[[-2.0921e+00,  9.9902e-33, -6.8406e-19,  4.5803e-41],
         [-6.8419e-19,  4.5803e-41,  1.9552e-04,  4.2477e+15],
         [-2.1203e-19,  4.5803e-41, -2.1213e-19,  4.5803e-41]],

        [[-5.8784e-05,  2.4339e-24, -2.1203e-19,  4.5803e-41],
         [ 1.3498e+14,  3.0680e-41,  9.9128e-23,  3.6741e-28],
         [-2.1170e-19,  4.5803e-41, -2.1283e-19,  4.5803e-41]]])
```

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

Tensor to Numpy, and Numpy to Tensor

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

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

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

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

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

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

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

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

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

Operations

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

The simplest operation is to add two tensors:

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

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

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

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

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

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

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

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

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

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

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

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

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

[14]:
```python
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 `a @ b`, similar to numpy.
- `torch.mm`: Performs the matrix product over two matrices, but doesn't support broadcasting (see documentation)
- `torch.bmm`: Performs the matrix product with a support batch dimension. If the first tensor T is of shape $(b \times n \times m)$, and the second tensor R $(b \times m \times p)$, the output O is of shape $(b \times n \times p)$, and has been calculated by performing b matrix multiplications of the submatrices of T and R: $O_i = T_i @ R_i$
- `torch.einsum`: Performs matrix multiplications and more (i.e. sums of products) using the Einstein summation convention. Explanation of the Einstein sum can be found in assignment 1.

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

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

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

[16]:
```python
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]:
```python
h = torch.matmul(x, W) # Verify the result by calculating it by hand too!
print("h", h)
```

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

Indexing

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

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

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

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

```
tensor([1, 5, 9])
```

```
[20]: print(x[0])      # First row
```

```
tensor([0, 1, 2, 3])
```

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

```
tensor([3, 7])
```

```
[22]: print(x[1:3, :]) # Middle two rows
```

```
tensor([[ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

Dynamic Computation Graph and Backpropagation

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

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

Given an input $\mathbf{x}$, 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 $\mathbf{x}$. We then define an **error measure** that tells us how wrong our network is; how bad it is in predicting output y from input $\mathbf{x}$. Based on this error measure, we can use the gradients to **update** the weights $\mathbf{W}$ that were responsible for the output, so that the next time we present input $\mathbf{x}$ to our network, the output will be closer to what we want.

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

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

```
False
```

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

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

```
True
```

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

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

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

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

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

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

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

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

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

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

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

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

[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:

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

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

GPU support

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

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

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

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

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

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

If you have a GPU on your computer but the command above returns False, make sure you have the correct CUDA-version installed. The `dl2021` environment comes with the CUDA-toolkit 11.3, which is selected for the Lisa supercomputer. Please change it if necessary (CUDA 11.1 is currently common on Colab). On Google Colab, make sure that you have selected a GPU in your runtime setup (in the menu, check under `Runtime -> Change runtime type`).

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

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

```
Device cuda
```

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

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

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

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

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

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

## GPU version
x = x.to(device)
# CUDA is asynchronous, so we need to use different timing functions
start = torch.cuda.Event(enable_timing=True)
end = torch.cuda.Event(enable_timing=True)
start.record()
_ = torch.matmul(x, x)
end.record()
torch.cuda.synchronize()  # Waits for everything to finish running on the GPU
print(f"GPU time: {0.001 * start.elapsed_time(end):6.5f}s")  # Milliseconds to seconds
```

```
CPU time: 0.20055s
GPU time: 0.00760s
```

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

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

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

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

3.1.2 Learning by example: Continuous XOR

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

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

The model

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

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

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

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

nn.Module

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

[36]:
```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:

[37]:
```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.

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

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

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

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

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

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

The data

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

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

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

The dataset class

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

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

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

(continues on next page)

(continued from previous page)

```
            size - Number of data points we want to generate
            std - Standard deviation of the noise (see generate_continuous_xor function)
        """
        super().__init__()
        self.size = size
        self.std = std
        self.generate_continuous_xor()

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

        self.data = data
        self.label = label

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

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

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

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

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

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

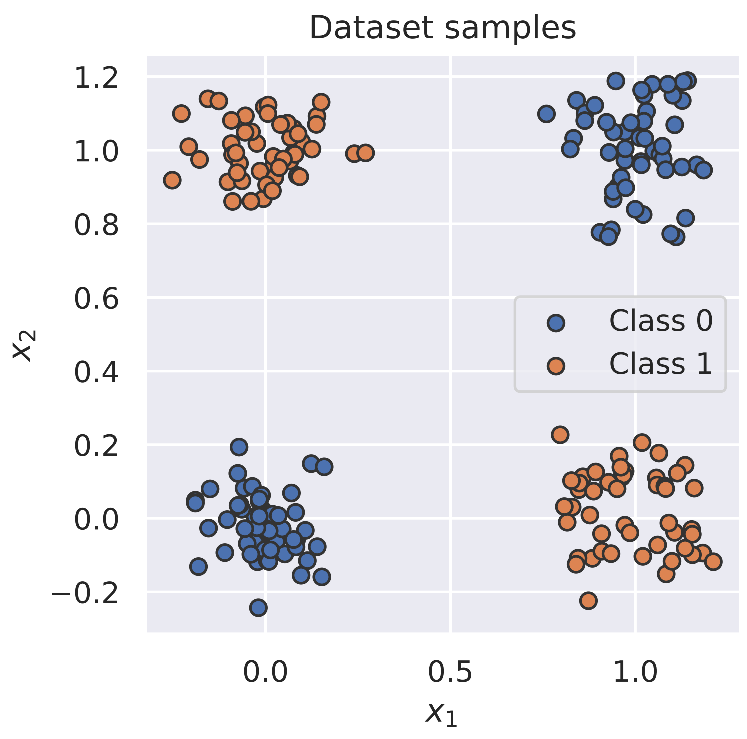
[43]:
```
def visualize_samples(data, label):
    if isinstance(data, torch.Tensor):
        data = data.cpu().numpy()
    if isinstance(label, torch.Tensor):
        label = label.cpu().numpy()
    data_0 = data[label == 0]
```

(continues on next page)

(continued from previous page)

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

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

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



The data loader class

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

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

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

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

Let's create a simple data loader below:

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

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

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

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

Optimization

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

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

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

Loss modules

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

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

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 adviced to use loss functions applied on “logits” where possible (remember to not apply a sigmoid on the output of the model in this case!). For our model defined above, we therefore use the module `nn.BCEWithLogitsLoss`.

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

Stochastic Gradient Descent

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

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

The optimizer provides two useful functions: `optimizer.step()`, and `optimizer.zero_grad()`. The step function updates the parameters based on the gradients as explained above. The function `optimizer.zero_grad()` sets the gradients of all parameters to zero. While this function seems less relevant at first, it is a crucial pre-step before performing backpropagation. If we 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.

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

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

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

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

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

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

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

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

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

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

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

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

[52]:
```
train_model(model, optimizer, train_data_loader, loss_module)
```

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

Saving a model

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

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

```
OrderedDict([('linear1.weight', tensor([[-2.6034, -3.3293],
        [ 1.9773, -2.4074],
        [-2.5968, -1.5909],
        [-0.5714, -0.8096]], device='cuda:0')), ('linear1.bias', tensor([ 1.4459, -1.
↪3992,  2.9883, -0.1376], device='cuda:0')), ('linear2.weight', tensor([[-4.4624,  3.
↪0882,  4.4031, -0.1372]], device='cuda:0')), ('linear2.bias', tensor([-1.6854], device=
↪'cuda:0'))])
```

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

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

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

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

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

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

```
Original model
 OrderedDict([('linear1.weight', tensor([[-2.6034, -3.3293],
        [ 1.9773, -2.4074],
        [-2.5968, -1.5909],
        [-0.5714, -0.8096]], device='cuda:0')), ('linear1.bias', tensor([ 1.4459, -1.
↪3992,  2.9883, -0.1376], device='cuda:0')), ('linear2.weight', tensor([[-4.4624,  3.
↪0882,  4.4031, -0.1372]], device='cuda:0')), ('linear2.bias', tensor([-1.6854], device=
↪'cuda:0'))])

Loaded model
 OrderedDict([('linear1.weight', tensor([[-2.6034, -3.3293],
        [ 1.9773, -2.4074],
        [-2.5968, -1.5909],
        [-0.5714, -0.8096]])), ('linear1.bias', tensor([ 1.4459, -1.3992,  2.9883, -0.
↪1376])), ('linear2.weight', tensor([[-4.4624,  3.0882,  4.4031, -0.1372]])), ('linear2.
↪bias', tensor([-1.6854]))])
```

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

Evaluation

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

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

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

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

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

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

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

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

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

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

    acc = true_preds / num_preds
    print(f"Accuracy of the model: {100.0*acc:4.2f}%")
```

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

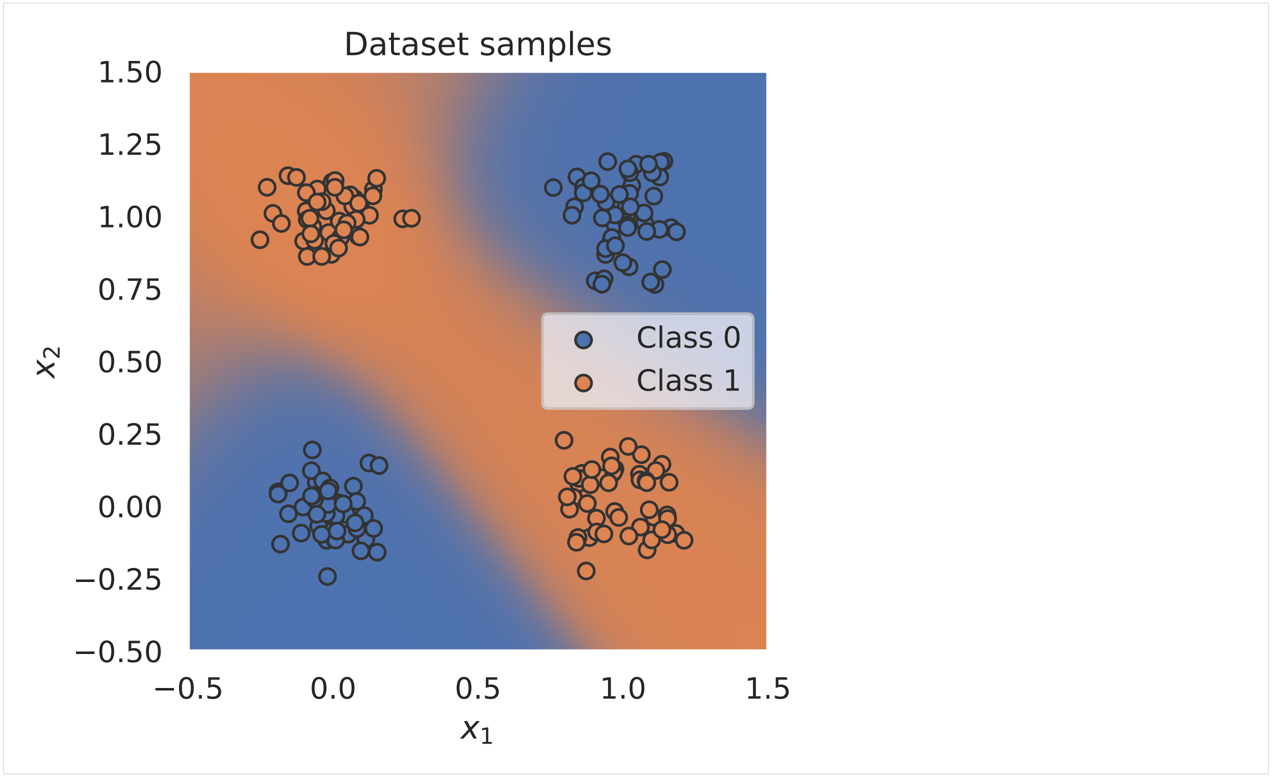
```
Accuracy of the model: 100.00%
```

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

Visualizing classification boundaries

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

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

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

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

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



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

3.1.3 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 comming series of Jupyter notebooks, we will discover more and more functionalities of PyTorch, so that you also get familiar to PyTorch concepts beyond the basics. If you are already interested in learning more of PyTorch, we recommend the official tutorial website that contains many tutorials on various topics. Especially logging with Tensorboard (official tutorial here) is a good practice. Hence, let's check it shortly out how we could use TensorBoard in our small example.

TensorBoard logging

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

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

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

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

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

```
[61]: def train_model_with_logger(model, optimizer, data_loader, loss_module, val_dataset, num_
↪epochs=100, logging_dir='runs/our_experiment'):
    # Create TensorBoard logger
    writer = SummaryWriter(logging_dir)
    model_plotted = False

    # Set model to train mode
    model.train()

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

    writer.close()
```

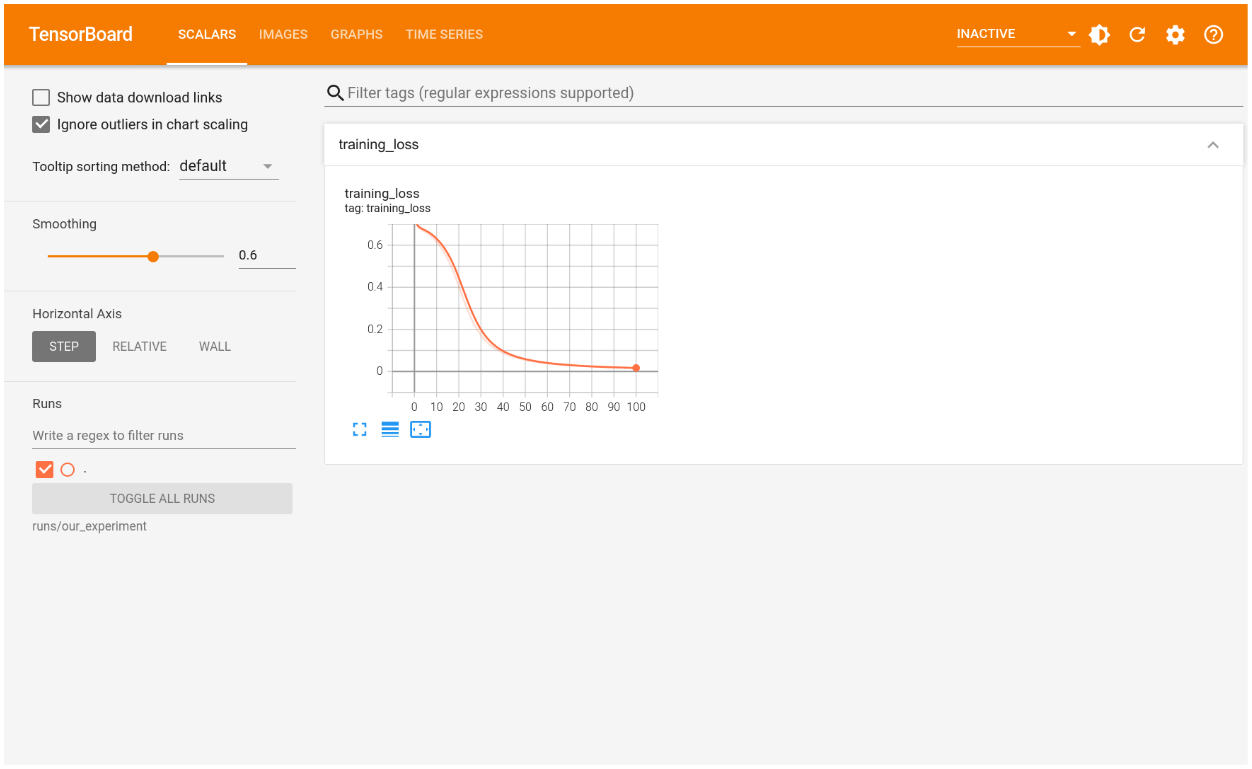
Let's use this method to train a model as before, with a new model and optimizer.

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

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

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

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



TensorBoard visualizations can help to identify possible issues with your model, and identify situations such as overfitting. You can also track the training progress while a model is training, since the logger automatically writes everything added to it to the logging file. Feel free to explore the TensorBoard functionalities further, and use it for your practicals.

3.2 Practical 1: Multi-Layer Perceptrons

Open notebook:
Authors: Phillip Lippe

In this practical, you will get a first experience in working with images and neural networks by implementing multi-layer perceptrons (MLPs). You will experiment with different optimization and regularization strategies to get the best performance out of your model, and investigate the limitations of using MLPs for image processing. This notebook is intended to guide you through the practical and you can edit it in any place. If you prefer working with standard python scripts, feel free to convert this notebook into a python script. To open this notebook on Google Colab, use the button above. Note that you need to copy this notebook into your own Google Drive to save the notebook and trained models. Otherwise, your progress will be lost when you close the browser tab.

First of all, let's start with importing some standard libraries:

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

(continues on next page)

(continued from previous page)

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

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

Throughout the practicals, we make use of several datasets and might want to save different models. For this, we define the two paths below. Adjust the paths as you like, but remember to keep them consistent across practicals to not download the dataset several times.

[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/practical1"
os.makedirs(CHECKPOINT_PATH, exist_ok=True)
```

Training deep neural networks is a stochastic process. Hence, every time you run the notebook, you naturally get different results. To obtain reproducible results, it is highly recommended to set a seed for all stochastic operations. Hence, let's define a `set_seed` function below.

[3]:
```
# 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.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
```

Finally, let's start with setting up the dataset we will use in this practical: CIFAR10. CIFAR10 is a very popular dataset for computer vision on low-resolution images (32x32 pixels). The task is to classify images into one of 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Our goal is to develop a neural network that can solve this task efficiently and accurately. To load the dataset, we can luckily make use of PyTorch's library `torchvision` which provides access to many popular vision datasets and more practical functions. So, let's load the dataset below:

[4]:
```
# Dataset statistics for normalizing the input values to zero mean and one std
DATA_MEANS = [0.491, 0.482, 0.447]
DATA_STD = [0.247, 0.243, 0.261]

# Transformations are applied on images when we want to access them. Here, we push the␣
↪images into a tensor
# and normalize the values. However, you can use more transformations, like␣
↪augmentations to prevent overfitting.
# Feel free to experiment with augmentations here once you have a first running MLP, but␣
↪remember to not apply
# any augmentations on the test data!
data_transforms = transforms.Compose([transforms.ToTensor(),
                                      transforms.Normalize(DATA_MEANS, DATA_STD)
                                     ])

# Loading the training dataset. We need to split it into a training and validation part
main_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=data_transforms,␣
↪download=True)
train_set, val_set = torch.utils.data.random_split(main_dataset, [45000, 5000],␣
↪generator=torch.Generator().manual_seed(42))

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

# Create data loaders for later
train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True,␣
↪pin_memory=True, num_workers=3)
val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False, drop_last=False,␣
↪num_workers=3)
test_loader = data.DataLoader(test_set, batch_size=128, shuffle=False, drop_last=False,␣
↪num_workers=3)
```

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

When working with data, it is always recommend to look at the data before blaming your model for not performing well if the data was incorrectly processed. Hence, let's plot the first 10 images of the CIFAR10 training dataset:

[5]:
```
NUM_IMAGES = 10
images = [train_set[idx][0] for idx in range(NUM_IMAGES)]
img_grid = torchvision.utils.make_grid(torch.stack(images, dim=0), nrow=5,␣
↪normalize=True, pad_value=0.5)
img_grid = img_grid.permute(1, 2, 0)
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(12,8))
plt.title("Dataset examples from CIFAR10", fontsize=20)
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

Now we are all set. So, let's dive into implementing our own MLP!

3.2.1 Part 1: Implementing the MLP

As a first step, let's implement a simple MLP.

MLP Module

You can make use of PyTorch's common functionalities, especially the `torch.nn` modules might be of help. The design choices of the MLP (e.g. the activation function) is left up to you, but for an initial setup, we recommend stacking linear layers with ReLU activation functions in between. Remember to not apply any activation function on the output.

```
[ ]: class MLP(nn.Module):

         def __init__(self, input_size=3072, num_classes=10, hidden_sizes=[256, 128]):
             """
             Inputs:
                 input_size - Size of the input images in pixels
                 num_classes - Number of classes we want to predict. The output size of the␣
↪MLP
                               should be num_classes.
                 hidden_sizes - A list of integers specifying the hidden layer sizes in the␣
↪MLP.
                                The MLP should have len(hidden_sizes)+1 linear layers.
```

(continues on next page)

(continued from previous page)

```
        """
        super().__init__()
        # TODO: Create the network based on the specified hidden sizes
        raise NotImplementedError

    def forward(self, x):
        # TODO: Apply the MLP on an input
        raise NotImplementedError
```

```
[ ]: # Let's test the MLP implementation
input_size = np.random.randint(low=64, high=3072)
num_classes = np.random.randint(low=5, high=20)
hidden_sizes = [np.random.randint(low=32, high=256) for _ in range(np.random.
↪randint(low=1, high=3))]
my_mlp = MLP(input_size=input_size, num_classes=num_classes, hidden_sizes=hidden_sizes)
my_mlp.to(device)
random_input = torch.randn(32, input_size, device=device)
random_output = my_mlp(random_input)
assert random_output.shape[0] == random_input.shape[0]
assert random_output.shape[1] == num_classes
```

Optimizer

To gain a better insight in the training of the neural networks, let's implement our own optimizer. 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 η. Usually, the learning rate can be seen as the "step size" of the update. A higher learning rate means that we change the weights more in the direction of the gradients, a smaller means we take shorter steps.

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

```
[ ]: 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
```

(continues on next page)

(continued from previous page)

```
        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 optimizer we are going to implement is the standard Stochastic Gradient Descent (SGD) with momentum. Plain SGD updates the parameters using the following equation:

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

The concept of momentum replaces the gradient in the update by an exponential average of all past gradients including the current one, which allows for a smoother training. The gradient update with momentum becomes:

$$\begin{aligned} m^{(t)} &= \beta_1 m^{(t-1)} + (1 - \beta_1) \cdot g^{(t)} \\ w^{(t)} &= w^{(t-1)} - \eta \cdot m^{(t)} \end{aligned}$$

Let's implement the optimizer below:

[ ]:
```
class SGDMomentum(OptimizerTemplate):

    def __init__(self, params, lr, momentum=0.9):
        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):
        # TODO: Implement the gradient update
        raise NotImplementedError
```

To verify that our optimizer is working, let's create a challenging surface over two parameter dimensions which we want to optimize to find the optimum:

[ ]:
```
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
```

[ ]:
```
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(figsize=(6, 6))
    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.)
    y = torch.arange(y_range[0], y_range[1], (y_range[1]-y_range[0])/100.)
    x, y = torch.meshgrid([x,y], indexing='ij')
```

(continues on next page)

(continued from previous page)

```
    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:

```
[ ]: def train_curve(optimizer_func, curve_func=pathological_curve_loss, num_updates=100,␣
↪init=[5,5]):
    """
    Inputs:
        optimizer_func - Constructor of the optimizer to use. Should only take a␣
↪parameter list
        curve_func - Loss function (e.g. pathological curvature)
        num_updates - Number of updates/steps to take when optimizing
        init - Initial values of parameters. Must be a list/tuple with two elements␣
↪representing w_1 and w_2
    Outputs:
        Numpy array of shape [num_updates, 3] with [t,:2] being the parameter values at␣
↪step t, and [t,2] the loss at t.
    """
    weights = nn.Parameter(torch.FloatTensor(init), requires_grad=True)
    optimizer = optimizer_func([weights])

    list_points = []
    for _ in range(num_updates):
        # TODO: Determine the loss for the current weights, save the weights and loss,␣
↪perform backpropagation
        raise NotImplementedError
```

(continues on next page)

(continued from previous page)

```
    points = torch.stack(list_points, dim=0).numpy()
    return points
```

Next, let's apply the optimizer 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.

```
[ ]: SGDMom_points = train_curve(lambda params: SGDMomentum(params, lr=10, momentum=0.9))
```

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.

```
[ ]: all_points = SGDMom_points
     ax = plot_curve(pathological_curve_loss,
                     x_range=(-np.absolute(all_points[:,0]).max(), np.absolute(all_points[:,
     ↪0]).max()),
                     y_range=(all_points[:,1].min(), all_points[:,1].max()),
                     plot_3d=False)
     ax.plot(SGDMom_points[:,0], SGDMom_points[:,1], color="red", marker="o", zorder=2, label=
     ↪"SGDMom")
     plt.legend()
     plt.show()
```

If the implementation is correct, you should see that the optimizer indeed reaches a point of very low w_2 ($w_2 < -7.5$) and $w_1 \approx 0$. If not, go back to your optimizer implementation and check what could go wrong.

Training and validation function

Now that we the MLP ready, the optimizer implemented, and the dataset loaded, we can look at implementing our own training function. The form of the training function is left mostly up to you, but we recommend that you test the model on the validation function after every 5 epochs, and save the best model. We provide a rough template below.

```
[ ]: def train_model(model, optimizer, train_data_loader, val_data_loader, num_epochs=25,␣
     ↪model_name="MyMLP"):
         # Set model to train mode
         model.to(device)
         best_val_acc = -1.0

         # Training loop
         for epoch in range(1, num_epochs+1):
             model.train()
             for imgs, labels in train_data_loader:
                 # TODO: Implement training loop with training on classification
                 raise NotImplementedError

             if epoch % 5 == 0:
                 # TODO: Evaluate the model and save if best
                 raise NotImplementedError

         # Load best model after training
         load_model(model, model_name)
```

```
[ ]: @torch.no_grad()
     def test_model(model, data_loader):
         # TODO: Test model and return accuracy
         raise NotImplementedError
```

```
[ ]: def save_model(model, model_name):
         # TODO: Save the parameters of the model
         raise NotImplementedError

     def load_model(model, model_name):
         # TODO: Load the parameters of the model
         raise NotImplementedError
```

```
[ ]: # TODO: Create model, optimizer, and start training
     raise NotImplementedError
```

```
[ ]: # TODO: Test best model on test set
     raise NotImplementedError
```

It is expected that you reach with the default configurations a validation and test accuracy of $\sim 52-53\%$. If you have reached this performance, we can consider this task as completed!

3.2.2 Part 2: Improving your MLP

Now that we have a basic MLP, let's try to improve over this default performance! Your task is to think about ways to maximize the performance of your MLP. Possible suggestions you can look at include:

- Do data augmentations help the model to generalize?
- Can regularization techniques (dropout, weight decay, etc.) help?
- Do deeper models perform better? Or is it better to have wide networks?
- Can normalization techniques (BatchNorm, LayerNorm, etc.) help?

Your task is to improve your model to reach at least 56% on the test set! But can you get even above this? Consider this as a challenge!

For this implementation, you can directly edit your model above. In your report, list the changes that you have made and discuss what affect they have. Further, repeat the experiment for *at least 3 seeds* to report stable improvements.

```
[ ]: # TODO: Improve the model
     raise NotImplementedError
```

3.2.3 Part 3: Investigating the limitations

Now that we have seen how we can optimize our MLP, it is good to investigate the limitations of the model as well. Images have a natural grid structure where close-by pixels are strongly related. Does the MLP make use of this structure? To investigate this question, we will run two experiments:

- Create a shuffle of pixels at the beginning of the training, and use the same shuffle throughout the training and validation.
- At each training and validation step, sample a new shuffle of pixels.

It is up to you whether you perform this investigation on the original plain MLP version or your optimized one. Implement a corresponding train and test function that support both variants of shuffling, and train two models.

```
[ ]: # TODO: Create a training and test function that supports the shuffling of pixels, both␣
     ↪fixed and newly generated for each batch
     raise NotImplementedError
```

```
[ ]: # TODO: Create model, optimizer, and start training on fixed shuffling of pixels
     raise NotImplementedError
```

```
[ ]: # TODO: Create model, optimizer, and start training on a new shuffling of pixels per␣
     ↪batch/image
     raise NotImplementedError
```

What results do you observe? What does this tell us about the MLP being aware of the image structure? Add your results and observations to your report.

3.2.4 Conclusion

You have reached the end of the practical, congratulations! Now, you should have a good idea of what it means to train a MLP, how we can use neural networks to perform image classification, and what tricks there are to improve a networks performance. In the next practicals, we will look at more advanced network structures beyond MLPs.

3.3 Practical 2: Convolutional Neural Networks

Open notebook:
Authors: Phillip Lippe

In this practical, we will implement 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 and learn how to implement some of them.

Let’s start with importing our standard libraries here.

```
[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
     import seaborn as sns
     sns.set()
```

(continues on next page)

(continued from previous page)

```
## 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

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

We will use the same the path variables DATASET_PATH and CHECKPOINT_PATH. Adjust the paths if necessary.

[2]:
```
# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/practical2"
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

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

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

Similarly as in the last practical, we use the CIFAR10 dataset and load it below:

[3]:
```
# Dataset statistics for normalizing the input values to zero mean and one std
DATA_MEANS = [0.491, 0.482, 0.447]
DATA_STD = [0.247, 0.243, 0.261]

test_transform = transforms.Compose([transforms.ToTensor(),
                                     transforms.Normalize(DATA_MEANS, DATA_STD)
                                     ])
# For training, we add some augmentation. Networks are too powerful and would overfit.
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                      transforms.RandomResizedCrop((32,32),scale=(0.8,1.
↪0),ratio=(0.9,1.1)),
                                      transforms.ToTensor(),
                                      transforms.Normalize(DATA_MEANS, DATA_STD)
                                     ])
# Loading the training dataset. We need to split it into a training and validation part
# We need to do a little trick because the validation set should not use the␣
↪augmentation.
train_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=train_transform,␣
↪download=True)
val_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=test_transform,␣
↪download=True)
```

(continues on next page)

(continued from previous page)

```
train_set, _ = torch.utils.data.random_split(train_dataset, [45000, 5000],␣
↪generator=torch.Generator().manual_seed(42))
_, val_set = torch.utils.data.random_split(val_dataset, [45000, 5000], generator=torch.
↪Generator().manual_seed(42))

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

# Create data loaders for later. Adjust batch size if you have a smaller GPU
train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True,␣
↪pin_memory=True, num_workers=3)
val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False, drop_last=False,␣
↪num_workers=3)
test_loader = data.DataLoader(test_set, batch_size=128, shuffle=False, drop_last=False,␣
↪num_workers=3)
```

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

3.3.1 PyTorch Lightning

In this practical and in many following ones, we will make use of the library PyTorch Lightning. PyTorch Lightning is a framework that simplifies your code needed to train, evaluate, and test a model in PyTorch. It also handles logging into TensorBoard, a visualization toolkit for ML experiments, and saving model checkpoints automatically with minimal code overhead from our side. This is extremely helpful for us as we want to focus on implementing different model architectures and spend little time on other code overhead. Note that at the time of writing/teaching, the framework has been released in version 1.6. Future versions might have a slightly changed interface and thus might not work perfectly with the code (we will try to keep it up-to-date as much as possible).

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

```
[4]: # PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by␣
↪default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.6
    import pytorch_lightning as pl
```

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

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

```
Global seed set to 42
```

```
[5]: 42
```

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

In PyTorch Lightning, we define `pl.LightningModule`'s (inheriting from `torch.nn.Module`) that organize our code into 5 main sections:

1. Initialization (`__init__`), where we create all necessary parameters/models
2. Optimizers (`configure_optimizers`) where we create the optimizers, learning rate scheduler, etc.
3. Training loop (`training_step`) where we only have to define the loss calculation for a single batch (the loop of optimizer.zero_grad(), loss.backward() and optimizer.step(), as well as any logging/saving operation, is done in the background)
4. Validation loop (`validation_step`) where similarly to the training, we only have to define what should happen per step
5. Test loop (`test_step`) which is the same as validation, only on a test set.

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

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

```
[6]: class CIFARModule(pl.LightningModule):

         def __init__(self, model_hparams, optimizer_hparams):
             """
             Inputs:
                 model_hparams - Hyperparameters for the model, as dictionary.
                 optimizer_hparams - Hyperparameters for the optimizer, as dictionary. This␣
     ↪includes learning rate, weight decay, etc.
             """
             super().__init__()
             # Exports the hyperparameters to a YAML file, and create "self.hparams" namespace
             self.save_hyperparameters()
             # Create model
             self.create_model()
             # Example input for visualizing the graph in Tensorboard
             self.example_input_array = torch.zeros((1, 3, 32, 32), dtype=torch.float32)

         def create_model(self):
             # No need to fill this yet, we will do it later in the notebook
             # Currently this function is a placeholder to create our model
             raise NotImplementedError

         def forward(self, imgs):
             # Forward function that is run when visualizing the graph
             return self.model(imgs)

         def configure_optimizers(self):
             # Create optimizer
             optimizer = optim.SGD(self.parameters(), **self.hparams.optimizer_hparams)
             # We will reduce the learning rate by 0.1 after 75 and 100 epochs
             scheduler = optim.lr_scheduler.MultiStepLR(
                 optimizer, milestones=[70, 90], 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
```

(continues on next page)

(continued from previous page)

```
        preds = self.model(imgs)
        loss = F.cross_entropy(preds, labels)
        acc = (preds.argmax(dim=-1) == labels).float().mean()

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

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

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

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

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

[7]:
```
# Callbacks
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
```

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:

[8]:
```
def train_model(save_name, data_loaders, max_epochs=100, **kwargs):
    """
```

(continues on next page)

(continued from previous page)

```
    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.
    """
    # 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
                         gpus=1 if str(device)=="cuda:0" else 0,                         ␣
↪                    # We run on a single GPU (if possible)
                         max_epochs=max_epochs,                                          ␣
↪                    # How many epochs to train for if no patience is set
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",␣
↪monitor="val_acc"),  # Save the best checkpoint based on the maximum val_acc recorded.␣
↪Saves only weights and not optimizer
                                    LearningRateMonitor("epoch")],                       ␣
↪                    # Log learning rate every epoch
                         check_val_every_n_epoch=10)                                     ␣
↪                   # Frequency with which we evaluate the model
    trainer.logger._log_graph = True         # If True, we plot the computation graph in␣
↪tensorboard
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't␣
↪need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, save_name + ".ckpt")
    if os.path.isfile(pretrained_filename):
        print(f"Found pretrained model at {pretrained_filename}, loading...")
        model = CIFARModule.load_from_checkpoint(pretrained_filename) # Automatically␣
↪loads the model with the saved hyperparameters
    else:
        pl.seed_everything(42) # To be reproducable
        model = CIFARModule(**kwargs)
        trainer.fit(model, data_loaders['train'], data_loaders['val'])
        model = CIFARModule.load_from_checkpoint(trainer.checkpoint_callback.best_model_
↪path) # Load best checkpoint after training

    # Test best model on validation and test set
    val_result = trainer.test(model, data_loaders['val'], verbose=False)
    test_result = trainer.test(model, data_loaders['test'], verbose=False)
    result = {"test": test_result[0]["test_acc"], "val": val_result[0]["test_acc"]}

    return model, result
```

This setup simplifies our training a lot, and we recommend using PyTorch Lightning for future practicals as well.

3.3.2 Part 1: ResNet

In the first part of the practical, you will implement a small 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 likes convolutions, activation functions, and normalizations). If we do backpropagation on such residual connections, we obtain:

$$\frac{\partial x_{l+1}}{\partial x_l} = \mathbf{I} + \frac{\partial F(x_l)}{\partial x_l}$$

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

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

For this practical, we will use the Pre-Activation 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 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. Let's try to implement the ResNet block below:

```
[ ]: class PreActResNetBlock(nn.Module):

         def __init__(self, c_in, subsample=False, c_out=-1):
             """
             Inputs:
                 c_in - Number of input features
                 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__()
             # TODO: Implement the network of the pre-activation ResNet block
             raise NotImplementedError

         def forward(self, x):
             # TODO: Implement the forward pass of the Pre-Activation ResNet block
             raise NotImplementedError
```

```
[ ]: # Testing the ResNet block
     c_in = np.random.randint(low=16, high=64)
     module = PreActResNetBlock(c_in=c_in, subsample=False)
```

(continues on next page)

(continued from previous page)

```
module.to(device)
img = torch.randn(4, c_in, 32, 32, device=device)
out = module(img)
for i in range(len(img.shape)):
    assert out.shape[i] == img.shape[i], f'Disagreement in shapes: output={out.shape},␣
↪input={img.shape}'

c_out = np.random.randint(low=16, high=64)
module = PreActResNetBlock(c_in=c_in, subsample=True, c_out=c_out)
module.to(device)
img = torch.randn(4, c_in, 32, 32, device=device)
out = module(img)
assert out.shape[1] == c_out
assert out.shape[2] == img.shape[2]//2
assert out.shape[3] == img.shape[3]//2
```

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

The three groups operate on the resolutions 32×32, 16×16 and 8×8 respectively. The blocks in orange denote ResNet blocks with downsampling. Additionally to these blocks, we have one initial convolution that maps the 3 color channels to the initial hidden channels, and a final linear layer after an average pooling over all features. Let's implement it below:

```
[ ]: class ResNet(nn.Module):

         def __init__(self, num_classes=10, num_blocks=[3,3,3], c_hidden=[16,24,32]):
             """
             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.
             """
             super().__init__()
             self.num_classes = num_classes
             self.num_blocks = num_blocks
             self.c_hidden = c_hidden
             self._create_network()
             self._init_params()

         def _create_network(self):
             # TODO: Create network with stacking blocks
             raise NotImplementedError

         def _init_params(self):
             # Fan-out focuses on the gradient distribution, and is commonly used in ResNets
             for m in self.modules():
```

(continues on next page)

(continued from previous page)

```
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        # TODO: Implement forward of ResNet
        raise NotImplementedError
```

[ ]:
```
# Testing the ResNet
num_classes = np.random.randint(low=5, high=20)
model = ResNet(num_classes=num_classes)
model.to(device)
img = torch.randn(4, 3, 32, 32, device=device)
out = model(img)
assert len(out.shape) == 2
assert out.shape[0] == img.shape[0]
assert out.shape[1] == num_classes
```

Now that we have all together, let's train the model. First, we need to define the `create_model` function in the `CIFARModule`:

[ ]:
```
def create_resnet(self):
    self.model = ResNet(num_classes=self.hparams.model_hparams['num_classes'],
                        num_blocks=self.hparams.model_hparams['num_blocks'],
                        c_hidden=self.hparams.model_hparams['c_hidden'])

CIFARModule.create_model = create_resnet
```

We call the training function below to start the training. We provide default parameters that should allow you to quickly train the model, but feel free to optimize the hyperparameters. For a final run in your report, please use 100 epochs.

[ ]:
```
resnet_model, resnet_results = train_model(save_name='ResNet',
                                           max_epochs=10,
                                           model_hparams={"num_classes": 10,
                                                          "c_hidden": [16,24,32],
                                                          "num_blocks": [2,2,3]},
                                           optimizer_hparams={"lr": 0.1,
                                                              "momentum": 0.9,
                                                              "weight_decay": 1e-4},
                                           data_loaders={"train": train_loader,
                                                         "val": val_loader,
                                                         "test": test_loader})
```

[ ]:
```
print(resnet_results)
```

Instead of just seeing the test result, it is recommended to also take a look at the TensorBoard log:

[ ]:
```
%load_ext tensorboard
%tensorboard --logdir ../saved_models/practical2/ResNet/
```

3.3.3 Part 2: Rotational Invariance

A common argument for CNNs over MLPs is that they have the inductive bias of being (approximately) shift invariant. If we move the image by one pixel to the left, most features remain unchanged and also just shift by one position. However, what about more complex transformations, like rotations? Your task in this part of the practical is to take the trained ResNet, and investigate how its test accuracy changes when you rotate the images. Create a plot of the validation accuracy over rotation angles (0 to 360 degree in steps of 10 degree).

[ ]:
```
@torch.no_grad()
def test_model_rotated(model, rotation_angle=0.0):
    # TODO: Evaluate model on images that have been rotated.
    # You might want to use torchvision.transforms.functional.rotate
    raise NotImplementedError
```

[ ]:
```
# TODO: Determine results
raise NotImplementedError
```

[ ]:
```
# TODO: Plot results
raise NotImplementedError
```

What do these results indicate? Add this plot in your report and discuss it.

3.3.4 Part 3: Pixel shuffling

In the first practical, we have investigated how the MLP reacts to shuffling the pixels of all images with a fixed, random permutation. Now, let's repeat this experiment for the CNNs. Does the CNN exploit the structural information differently than the MLP?

The first step is to create datasets with a new shuffling of pixels:

[ ]:
```
# TODO: Create datasets and data loaders with a fixed, random shuffle of pixels
raise NotImplementedError
```

Next, we can start the training of the model. You can limit your number of epochs to a smaller number like 10.

[ ]:
```
# TODO: Create datasets and data loaders with a fixed, random shuffle of pixels
raise NotImplementedError
```

[ ]:
```
# TODO: Print the results and look at your tensorboard
raise NotImplementedError
```

Add your observations to your report and discuss what this implies for CNNs.

3.3.5 Conclusion

In this practical, you gained some handson experience with CNNs on computer vision for classification. Your experiments probably give you a good indication why CNNs are a very popular choice for computer vision problems, but also what they can and cannot do.

3.4 Practical 3: Vision Transformers

Open notebook:
Authors: Phillip Lippe

In this practical, we will take a closer look at a recent new trend: Transformers for Computer Vision. Since Alexey Dosovitskiy et al. successfully applied a Transformer on a variety of image recognition benchmarks, there have been an incredible amount of follow-up works showing that CNNs might not be optimal architecture for Computer Vision anymore. But how do Vision Transformers work exactly, and what benefits and drawbacks do they offer in contrast to CNNs? We will answer these questions by implementing a Vision Transformer ourselves and train it on the popular, small dataset CIFAR10.

Let's start with importing our standard set of libraries.

[1]:
```
## Standard libraries
import os
import numpy as np
import random
import math
import json
from functools import partial
from PIL import Image

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
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 CIFAR10
from torchvision import transforms

# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by␣
↪default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.6
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
```
(continues on next page)

(continued from previous page)

```
# Import tensorboard
%load_ext tensorboard

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/practical3"

# 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

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

```
Global seed set to 42
```

```
Device: cuda:0
```

We load the CIFAR10 dataset below.

[2]:
```
# Dataset statistics for normalizing the input values to zero mean and one std
DATA_MEANS = [0.491, 0.482, 0.447]
DATA_STD = [0.247, 0.243, 0.261]

test_transform = transforms.Compose([transforms.ToTensor(),
                                     transforms.Normalize(DATA_MEANS, DATA_STD)
                                     ])
# For training, we add some augmentation. Networks are too powerful and would overfit.
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                      transforms.RandomResizedCrop((32,32),scale=(0.8,1.
↪0),ratio=(0.9,1.1)),
                                      transforms.ToTensor(),
                                      transforms.Normalize(DATA_MEANS, DATA_STD)
                                     ])
# Loading the training dataset. We need to split it into a training and validation part
# We need to do a little trick because the validation set should not use the␣
↪augmentation.
train_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=train_transform,␣
↪download=True)
val_dataset = CIFAR10(root=DATASET_PATH, train=True, transform=test_transform,␣
↪download=True)
train_set, _ = torch.utils.data.random_split(train_dataset, [45000, 5000],␣
↪generator=torch.Generator().manual_seed(42))
_, val_set = torch.utils.data.random_split(val_dataset, [45000, 5000], generator=torch.
↪Generator().manual_seed(42))

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

(continues on next page)

(continued from previous page)

```
# Create data loaders for later. Adjust batch size if you have a smaller GPU
train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True,␣
↪pin_memory=True, num_workers=3)
val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False, drop_last=False,␣
↪num_workers=3)
test_loader = data.DataLoader(test_set, batch_size=128, shuffle=False, drop_last=False,␣
↪num_workers=3)
```

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

```
[3]: # For later, we keep a set of example images
CIFAR_images = torch.stack([val_set[idx][0] for idx in range(4)], dim=0)
img_grid = torchvision.utils.make_grid(CIFAR_images, nrow=4, normalize=True, pad_value=0.
↪9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(12,8))
plt.title("Image examples of the CIFAR10 dataset", fontsize=20)
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

3.4.1 Part 1: Building a Transformer for image classification

Transformers have been originally proposed to process sets since it is a permutation-equivariant architecture, i.e., producing the same output permuted if the input is permuted. To apply Transformers to sequences, one commonly adds a positional encoding to the input feature vectors, and the model learns by itself what to do with it. So, why not do the same thing on images? This is exactly what Alexey Dosovitskiy et al. proposed in their paper "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". Specifically, the Vision Transformer is a model for image classification that views images as sequences of smaller patches. As a preprocessing step, we split an image of, for example, 48×48 pixels into 9 16×16 patches. Each of those patches is considered to be a "word"/"token" and projected to a feature space. With adding positional encodings and a token for classification on top, we can apply a Transformer as usual to this sequence and start training it for our task. A nice GIF visualization of the architecture is shown below (figure credit - Phil Wang):

We will walk step by step through the Vision Transformer, and implement all parts by ourselves. First, let's implement the image preprocessing: an image of size $N \times N$ has to be split into $(N/M)^2$ patches of size $M \times M$. These represent the input words to the Transformer. Implement this logic in the function below:

```
[ ]: def img_to_patch(x, patch_size, flatten_channels=True):
         """
         Inputs:
             x - torch.Tensor representing the image of shape [B, C, H, W]
             patch_size - Number of pixels per dimension of the patches (integer)
             flatten_channels - If True, the patches will be returned in a flattened format
                                as a feature vector instead of a image grid.
         Outputs:
             Tensor of shape [B, H*W/patch_size^2, C, patch_size, patch_size] if flatten_
     ↪channels=False,
             and [B, H*W/patch_size^2, C*patch_size^2] otherwise.
         """
         # TODO: Implement the splitting of images into patches
         raise NotImplementedError
```

```
[ ]: imgs = torch.randn(4, 3, 48, 48)
     patch_size = 16
     out = img_to_patch(imgs, patch_size, flatten_channels=False)
     assert len(out.shape) == 5
     assert out.shape[0] == imgs.shape[0]
     assert out.shape[1] == imgs.shape[2]*imgs.shape[3]/patch_size**2
     assert out.shape[2] == imgs.shape[1]
     assert out.shape[3] == patch_size
     assert out.shape[4] == patch_size

     out = img_to_patch(imgs, patch_size, flatten_channels=True)
     assert len(out.shape) == 3
     assert out.shape[0] == imgs.shape[0]
     assert out.shape[1] == imgs.shape[2]*imgs.shape[3]/patch_size**2
     assert out.shape[2] == imgs.shape[1]*patch_size**2
```

Let's take a look at how that works for our CIFAR examples above. For our images of size 32×32, we choose a patch size of 4. Hence, we obtain sequences of 64 patches of size 4×4. We visualize them below:

```
[ ]: img_patches = img_to_patch(CIFAR_images, patch_size=4, flatten_channels=False)

     fig, ax = plt.subplots(CIFAR_images.shape[0], 1, figsize=(14,3))
     fig.suptitle("Images as input sequences of patches")
     for i in range(CIFAR_images.shape[0]):
         img_grid = torchvision.utils.make_grid(img_patches[i], nrow=64, normalize=True, pad_
     ↪value=0.9)
         img_grid = img_grid.permute(1, 2, 0)
         ax[i].imshow(img_grid)
         ax[i].axis('off')
     plt.show()
     plt.close()
```

Compared to the original images, it is much harder to recognize the objects from those patch lists now. Still, this is the input we provide to the Transformer for classifying the images. The model has to learn itself how it has to combine the patches to recognize the objects. The inductive bias in CNNs that an image is a grid of pixels, is lost in this input format.

After we have looked at the preprocessing, we can now start building the Transformer model. You can make use of the PyTorch module `nn.MultiheadAttention` (docs) here. Further, we use the Pre-Layer Normalization version of the Transformer blocks proposed by Ruibin Xiong et al. in 2020. The idea is to apply Layer Normalization not in between residual blocks, but instead as a first layer in the residual blocks. This reorganization of the layers supports better gradient flow and removes the necessity of a warm-up stage. A visualization of the difference between the standard Post-LN and the Pre-LN version is shown below.

First, implement a Pre-LN attention block below:

```
[ ]: class AttentionBlock(nn.Module):

         def __init__(self, embed_dim, hidden_dim, num_heads, dropout=0.0):
             """
             Inputs:
                 embed_dim - Dimensionality of input and attention feature vectors
                 hidden_dim - Dimensionality of hidden layer in feed-forward network
                              (usually 2-4x larger than embed_dim)
                 num_heads - Number of heads to use in the Multi-Head Attention block
                 dropout - Amount of dropout to apply in the feed-forward network
             """
             super().__init__()
             # TODO: Implement an pre-LN attention block
             raise NotImplementedError

         def forward(self, x):
             """
             Inputs:
                 x - Input tensor of shape [Num Patches, Batch size, embed_dim]
             """
             # TODO: Implement the forward pass
             raise NotImplementedError
```

```
[ ]: # Testing the attention block
     num_heads = np.random.randint(low=4, high=16)
     embed_dim = num_heads * np.random.randint(low=16, high=32)
     hidden_dim = np.random.randint(low=128, high=512)
     block = AttentionBlock(embed_dim, hidden_dim, num_heads)
     block.to(device)
     inp = torch.randn(10, 32, embed_dim, device=device)
     out = block(inp)
     for i in range(len(inp.shape)):
         assert out.shape[i] == inp.shape[i]

     # Checking whether batch and patch dimension are correct
     inp2 = inp.clone()
     inp2[:,0] = 0.0
     out2 = block(inp2)
     diff = (out - out2).abs()
     assert (diff[:,0] > 1e-4).any(), 'Output tensor shows no difference although input has␣
     ↪changed'
     assert (diff[:,1:] < 1e-4).all(), 'Other tensors besides the changed batch element have␣
     ↪altered outputs. Check the dimensions'
```

(continues on next page)

(continued from previous page)

Now we have all modules ready to build our own Vision Transformer. Besides the Transformer encoder, we need the following modules:

- A **linear projection** layer that maps the input patches to a feature vector of larger size. It is implemented by a simple linear layer that takes each $M \times M$ patch independently as input.
- A **classification token** that is added to the input sequence. We will use the output feature vector of the classification token (CLS token in short) for determining the classification prediction.
- Learnable **positional encodings** that are added to the tokens before being processed by the Transformer. Those are needed to learn position-dependent information, and convert the set to a sequence. Since we usually work with a fixed resolution, we can learn the positional encodings instead of having the pattern of sine and cosine functions.
- An **MLP head** that takes the output feature vector of the CLS token, and maps it to a classification prediction. This is usually implemented by a small feed-forward network or even a single linear layer.

With those components in mind, let's implement the full Vision Transformer below:

```
[ ]: class VisionTransformer(nn.Module):

         def __init__(self, embed_dim, hidden_dim, num_channels, num_heads, num_layers, num_
     ↪classes, patch_size, num_patches, dropout=0.0):
             """
             Inputs:
                 embed_dim - Dimensionality of the input feature vectors to the Transformer
                 hidden_dim - Dimensionality of the hidden layer in the feed-forward networks
                              within the Transformer
                 num_channels - Number of channels of the input (3 for RGB)
                 num_heads - Number of heads to use in the Multi-Head Attention block
                 num_layers - Number of layers to use in the Transformer
                 num_classes - Number of classes to predict
                 patch_size - Number of pixels that the patches have per dimension
                 num_patches - Maximum number of patches an image can have
                 dropout - Amount of dropout to apply in the feed-forward network and
                           on the input encoding
             """
             super().__init__()
             # TODO: Implement all elements of the full Vision Transform
             raise NotImplementedError

         def forward(self, x):
             # TODO: Implement the forward pass
             raise NotImplementedError
```

```
[ ]: # Testing the Vision Transformer module
     num_heads = np.random.randint(low=4, high=16)
     embed_dim = num_heads * np.random.randint(low=16, high=32)
     hidden_dim = np.random.randint(low=128, high=512)
     num_channels = 3
     num_layers = np.random.randint(low=2, high=4)
     num_classes = np.random.randint(low=5, high=20)
     patch_size = [2,4,8][np.random.randint(low=0, high=3)]
```

(continues on next page)

(continued from previous page)

```
num_patches = int((32/patch_size)**2)

vit_module = VisionTransformer(embed_dim, hidden_dim, num_channels, num_heads, num_
↪layers, num_classes, patch_size, num_patches)
vit_module.to(device)
imgs = torch.randn(4, 3, 32, 32, device=device)
out = vit_module(imgs)
assert out.shape[0] == imgs.shape[0]
assert out.shape[1] == num_classes

# Checking whether batch and patch dimension are correct
imgs2 = imgs.clone()
imgs2[0] = 0.0
out2 = vit_module(imgs2)
diff = (out2 - out).abs()
assert (diff[0] > 1e-4).any(), 'Output tensor shows no difference although input has␣
↪changed'
assert (diff[1:] < 1e-4).all(), 'Other tensors besides the changed batch element have␣
↪altered outputs. Check the dimensions'
```

Finally, we can put everything into a PyTorch Lightning Module as usual. We use `torch.optim.AdamW` as the optimizer, which is Adam with a corrected weight decay implementation. Since we use the Pre-LN Transformer version, we do not need to use a learning rate warmup stage anymore.

[ ]:
```
class ViT(pl.LightningModule):

    def __init__(self, model_kwargs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.model = VisionTransformer(**model_kwargs)
        self.example_input_array = next(iter(train_loader))[0]

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

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(), lr=self.hparams.lr, weight_decay=1e-3)
        lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[75, 90],␣
↪gamma=0.1)
        return [optimizer], [lr_scheduler]

    def _calculate_loss(self, batch, mode="train"):
        # TODO: Implement step to calculate the loss and accuracy for a batch
        raise NotImplementedError

    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")
```

(continues on next page)

(continued from previous page)

```
    def test_step(self, batch, batch_idx):
        self._calculate_loss(batch, mode="test")
```

3.4.2 Experiments

Commonly, Vision Transformers are applied to large-scale image classification benchmarks such as ImageNet to leverage their full potential. However, here we take a step back and ask: can Vision Transformer also succeed on classical, small benchmarks such as CIFAR10? To find this out, we train a Vision Transformer from scratch on the CIFAR10 dataset. Let's first create a training function for our PyTorch Lightning module which also loads the pre-trained model if you have downloaded it above.

```
[ ]: def train_model(max_epochs=100, **kwargs):
         trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "ViT"),
                              gpus=1 if str(device)=="cuda:0" else 0,
                              max_epochs=max_epochs,
                              callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",␣
     ↪monitor="val_acc"),
                                         LearningRateMonitor("epoch")],
                              check_val_every_n_epoch=10)
         trainer.logger._log_graph = True         # If True, we plot the computation graph in␣
     ↪tensorboard
         trainer.logger._default_hp_metric = None # Optional logging argument that we don't␣
     ↪need

         # Check whether pretrained model exists. If yes, load it and skip training
         pretrained_filename = os.path.join(CHECKPOINT_PATH, "ViT.ckpt")
         if os.path.isfile(pretrained_filename):
             print(f"Found pretrained model at {pretrained_filename}, loading...")
             model = ViT.load_from_checkpoint(pretrained_filename) # Automatically loads the␣
     ↪model with the saved hyperparameters
         else:
             pl.seed_everything(42) # To be reproducable
             model = ViT(**kwargs)
             trainer.fit(model, train_loader, val_loader)
             model = ViT.load_from_checkpoint(trainer.checkpoint_callback.best_model_path) #␣
     ↪Load best checkpoint after training

         # Test best model on validation and test set
         val_result = trainer.test(model, val_loader, verbose=False)
         test_result = trainer.test(model, test_loader, verbose=False)
         result = {"test": test_result[0]["test_acc"], "val": val_result[0]["test_acc"]}

         return model, result
```

Now, we can already start training our model. As seen in our implementation, we have a couple of hyperparameters that we have to set. When creating this notebook, we have performed a small grid search over hyperparameters and listed the best hyperparameters in the cell below. Nevertheless, it is worth discussing the influence that each hyperparameter has, and what intuition we have for choosing its value.

First, let's consider the patch size. The smaller we make the patches, the longer the input sequences to the Transformer become. While in general, this allows the Transformer to model more complex functions, it requires a longer computation time due to its quadratic memory usage in the attention layer. Furthermore, small patches can make the task more difficult since the Transformer has to learn which patches are close-by, and which are far away. We experimented with patch sizes of 2, 4, and 8 which gives us the input sequence lengths of 256, 64, and 16 respectively. We found 4 to result in the best performance and hence pick it below.

Next, the embedding and hidden dimensionality have a similar impact on a Transformer as to an MLP. The larger the sizes, the more complex the model becomes, and the longer it takes to train. In Transformers, however, we have one more aspect to consider: the query-key sizes in the Multi-Head Attention layers. Each key has the feature dimensionality of `embed_dim/num_heads`. Considering that we have an input sequence length of 64, a minimum reasonable size for the key vectors is 16 or 32. Lower dimensionalities can restrain the possible attention maps too much. To reduce the computational complexity, we recommend using 4 heads, 128 embedding dimensionality and 256 hidden dimensionality for a start.

Finally, the learning rate for Transformers is usually relatively small, and in papers, a common value to use is 3e-5. However, since we work with a smaller dataset and have a potentially easier task, we found that we are able to increase the learning rate to 3e-4 without any problems.

Feel free to explore the hyperparameters yourself by changing the values below. For a final run for the report, increase the epochs to 100.

```
[ ]: model, results = train_model(model_kwargs={
                                     'embed_dim': 128,
                                     'hidden_dim': 256,
                                     'num_heads': 8,
                                     'num_layers': 6,
                                     'patch_size': 8,
                                     'num_channels': 3,
                                     'num_patches': 16,
                                     'num_classes': 10,
                                     'dropout': 0.0
                                   },
                                   lr=3e-4,
                                   max_epochs=10)
     print("ViT results", results)
```

```
[ ]: # Opens tensorboard in notebook. Adjust the path to your CHECKPOINT_PATH!
     %tensorboard --logdir ../saved_models/practical3/ViT
```

What accuracy does the ViT achieve? How does this compare to the CNN you have implemented in the second assignment? Add the plots of the validation accuracy and a discussion of the previous questions in your report.

3.4.3 Bonus 1: Importance of Positional Embeddings

This part of the practical is not mandatory, but we recommend going through it if you have time. Here, we are interested in what elements of a Transformer are crucial. One considerable difference to CNNs is that Transformers look at the images as patches instead of a full grid, which removes some inductive biases about the positional information. To still have access to positional information, we use position embeddings. However, how important are those to the Transformer actually? Do we see a noticable accuracy drop if we don't use the position embeddings? Or is it looking at the images as a big bag of words anyways? Finally, how does this relation change when using different patch sizes? These questions you should try to find an answer in this part of the practical. Specifically, train a Transformer without positional embeddings, and compare it to your original Transformer. Repeat the experiment for a smaller patch size (4 or even 2) and compare how the accuracies have changed.

3.4.4 Conclusion

In this tutorial, we have implemented our own Vision Transformer from scratch and applied it to the task of image classification. Vision Transformers work by splitting an image into a sequence of smaller patches, use those as input to a standard Transformer encoder. While Vision Transformers achieved outstanding results on large-scale image recognition benchmarks such as ImageNet, they considerably underperform when being trained from scratch on small-scale datasets like CIFAR10. The reason is that in contrast to CNNs, Transformers do not have the inductive biases of translation invariance and the feature hierarchy (i.e. larger patterns consist of many smaller patterns). However, these aspects can be learned when enough data is provided, or the model has been pre-trained on other large-scale tasks. Considering that Vision Transformers have just been proposed end of 2020, there is likely a lot more to come on Transformers for Computer Vision.

References

Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." International Conference on Representation Learning (2021). link

Chen, Xiangning, et al. "When Vision Transformers Outperform ResNets without Pretraining or Strong Data Augmentations." arXiv preprint arXiv:2106.01548 (2021). link

Tolstikhin, Ilya, et al. "MLP-mixer: An all-MLP Architecture for Vision." arXiv preprint arXiv:2105.01601 (2021). link

Xiong, Ruibin, et al. "On layer normalization in the transformer architecture." International Conference on Machine Learning. PMLR, 2020. link

3.5 Practical 4: Regular Group Convolutions

Open notebook:
Authors: David Knigge

3.5.1 Introduction

In this notebook, we will be implementing regular group convolutional networks from scratch, only making use of `pytorch` primitives. The goal is to get familiar with the practical considerations to take into account when actually implementing these convolutional networks.

You will be asked to fill in some gaps yourself. The pieces of code you are expected to fill in are surrounded by demarcations, like so:

```
a = 1
b = 2

# We calculate c = a + b

### YOUR CODE STARTS HERE ###
c = ...
### AND ENDS HERE ###
```

And you'd be expected to fill in something along the lines of `c = a + b` for `c = ...`. We've included some assertions to test for correctness of resulting tensor shapes.

We've also scattered some questions throughout the notebook to test your understanding of the implementation and concepts used. Please include your answers to these questions (and any other observations you deem relevant!) in your report.

If you'd like a refresher of the lecture, here we give a brief overview of the operations we are going to work with / implement. These will be treated more extensively below. For simplicity of notation, here we assume each CNN layer consists of only a single channel. Questions and feedback may be forwarded to David Knigge; d.m.knigge@uva.nl.

Brief recap on CNNs

Conventional CNNs make use of the convolution operator, here defined over $\mathbb{R}^2$ for a signal $f : \mathbb{R}^2 \to \mathbb{R}$ and a kernel $k : \mathbb{R}^2 \to \mathbb{R}$ at $\mathbf{x} \in \mathbb{R}^2$:

$$(f * k)(\mathbf{x}) = \int_{\mathbb{R}^2} f(\tilde{\mathbf{x}})k(\tilde{\mathbf{x}} - \mathbf{x})\mathrm{d}\tilde{\mathbf{x}},$$

As we can see, the convolution operation comes down to an inner product of the function f and a shifted kernel k.

Sidenote: In reality CNNs implement a discretised version of this operation;

$$\begin{aligned}(f * k)(\mathbf{x}) &= \sum_{\tilde{\mathbf{x}} \in \mathbb{Z}^2} f(\tilde{\mathbf{x}})k(\mathbf{x} - \tilde{\mathbf{x}})\Delta\tilde{\mathbf{x}} \\ &= \sum_{\tilde{\mathbf{x}} \in \mathbb{Z}^2} f(\tilde{\mathbf{x}})k(\mathbf{x} - \tilde{\mathbf{x}})\end{aligned}$$

Where above, since pixels in an image are generally evenly spaced, we set $\Delta\tilde{\mathbf{x}} = 1$. For this recap we stay in the continuous domain for simplicity.

In convolution layers, like PyTorch's `Conv2D` implementation, the above operation is carried out for every $\mathbf{x} \in \mathbb{Z}^2$ (limited of course to the domain over which the image is defined). Because the same set of weights is used throughout the input, the output of this operation is equivariant to transformations from the translation group $\mathbb{R}^2$. Furthermore f, k usually consist of a number of channels, which are all summed over.

In this tutorial, we will use PyTorch's `torch.nn.functional.conv2d()` function to perform this integration operation at every position in input feature map. This saves us having to implement the convolution operation ourselves.

Brief recap on GCNNs

In regular group convolutions, the goal is to have a CNN of which are not only equivariant to translations $\mathbb{R}^2$, but which are also equivariant to another (usually broader) group of interest G. We focus specifically on groups which are combinations of translations $\mathbb{R}^2$ and some group of interest H. In this tutorial we keep to the group of 90 degree rotations in 2D; the Cyclic group of order 4 $H = C_4$.

We will operate on 2D images, generally defined on $\mathbb{R}^2$, as such, the first step in constructing a network which can track under which *pose* (read: transformation from a group $G = \mathbb{R}^2 \rtimes H$) a feature in the input occurs, we need to transfer our signal to a domain in which the same feature under a different pose is disentangled. This happens through *the lifting convolution*, which maps features in our input signal $f_{in} : \mathbb{R}^2 \to \mathbb{R}$ to a feature map on the group $f_{out} : G \to \mathbb{R}$. For a signal and kernel f, k both defined on $\mathbb{R}^2$, and a group element $g = (\boldsymbol{x}, h) \in G = \mathbb{R}^2 \rtimes H$:

$$(f *_{\text{lifting}} k)(g) = \int_{\mathbb{R}^2} f(\tilde{\mathbf{x}})k_h(\tilde{\mathbf{x}} - \mathbf{x})\,\mathrm{d}\tilde{\mathbf{x}}.$$

Where k_h is the kernel $k : \mathbb{R}^2 \to \mathbb{R}$ transformed under the regular representation $\mathcal{L}_h$ of a group element $h \in H$; $k_h = \frac{1}{|h|}\mathcal{L}_h[k]$.

Sidenote: The factor $\frac{1}{|h|}$, with $|h|$ the determinant of the matrix representation of h in $\mathbb{R}^2$, accounts for a possible change in volume on $\mathbb{R}^2$ that h might have. Working with the cyclic group, we don't encounter this problem (the determinant of a rotation matrix is 1, volumes are invariant to rotations on $\mathbb{R}^2$), but if you'd like to implement equivariance to for example the dilation group, this becomes important.

Next, now that we have a feature map defined on the group; $f_{out} : G \rightarrow \mathbb{R}$, we apply group convolutions, extending the convolution operation to an integral over the entire group G;

$$\begin{aligned}(f *_{\text{group}} k)(g) &= \int_G f(\tilde{g})k(g^{-1}\cdot\tilde{g})\mathrm{d}\tilde{g}\\ &= \int_{\mathbb{R}^2}\int_H f(\tilde{\mathbf{x}},\tilde{h})\mathcal{L}_x\mathcal{L}_h k(\tilde{\mathbf{x}},\tilde{h})\frac{1}{|h|}\,\mathrm{d}\tilde{\mathbf{x}}\,\mathrm{d}\tilde{h}\\ &= \int_{\mathbb{R}^2}\int_H f(\tilde{\mathbf{x}},\tilde{h})k(h^{-1}(\tilde{\mathbf{x}}-\mathbf{x}),h^{-1}\cdot\tilde{h})\frac{1}{|h|}\,\mathrm{d}\tilde{\mathbf{x}}\,\mathrm{d}\tilde{h}.\end{aligned}$$

The main difference with the lifting convolution is that the signal and kernel f, k are both functions on $G;G \rightarrow \mathbb{R}$, and the integral reflects this by extending over the entire group G. Other than that, there is little difference!

After a number of such group convolutional layers, we will want to ultimately obtain a representation that is invariant to the group action. We can do this by performing a projection which collapses our function defined over G to a single point, with an operation that is invariant to the group action (summing, averaging, max, min).

After this short refresher, let's get to coding!

Installing and importing some useful packages

Here we install and import some libraries that we will use throughout this tutorial. We use the `pytorch` as our deep learning framework of choice. Note that for ease of model training and tracking, we additionally make use of `pytorch-lightning`.

[1]:
```
## Standard libraries
import os
import numpy as np
import math
from PIL import Image
from types import SimpleNamespace
from functools import partial

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0

## PyTorch
import torch
import torch.nn as nn
import torch.utils.data as data
import torch.optim as optim
## Torchvision
import torchvision
from torchvision.datasets import MNIST
from torchvision import transforms
```

(continues on next page)

(continued from previous page)

```
## PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by␣
↪default. Hence, we do it here if necessary
    !pip3 install pytorch-lightning>=1.6 --quiet
    import pytorch_lightning as pl
import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
```

[2]:
```
# Path to the folder where the datasets are be downloaded (e.g. MNIST)
DATASET_PATH = "../data"
os.makedirs(DATASET_PATH, exist_ok=True)
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/practical4"
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

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

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

As an example image, we will use the popular image of paprikas. If you are working on GoogleColab, you need to download this image, which we will do below:

[3]:
```
import urllib.request
from urllib.error import HTTPError
# Github URL where the image is stored
base_url = "https://raw.githubusercontent.com/phlippe/asci_cbl_practicals/main/assets/"
# Files to download
files = ["paprika.tiff"]

# For each file, check whether it already exists. If not, try downloading it.
for file_name in files:
    file_path = os.path.join(DATASET_PATH, file_name)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file manually from␣
↪the Github, or ask one of your TAs with the following error message:\n", e)
```

3.5.2 Part 1: Group theory

1.1 What is a group?

To start off, we recap some of the group theoretical preliminaries. Recall that a group is defined by a tuple $(G, \cdot)$, where G is a set of group elements and $\cdot$ the binary group action which tells us how elements $g \in G$ combine. The group action $\cdot$ needs to satisfy:

1. Closure. G is closed under $\cdot$; for all $g_1, g_2 \in G$ we have $g_1 \cdot g_2 \in G$.
2. Identity. There exists an identity element e s.t. for each $g \in G$, we have $e \cdot g = g \cdot e = g$.
3. Inverse. For every element $g \in G$ we have an element $g^{-1} \in G$, s.t. $g \cdot g^{-1} = e$.
4. Associativity. For every set of elements $g_1, g_2, g_3 \in G$, we have $(g_1 \cdot g_2) \cdot g_3 = g_1 \cdot (g_2 \cdot g_3)$.

The group can have an action on functions defined on $\mathbb{R}^2$, which we can instantiate through the *regular representation* $\mathcal{L}_g^{\mathbb{G}\rightarrow\mathbb{R}^2}$. For simplicity, we write $\mathcal{L}_g$. It is given by:

$$\mathcal{L}_g f(\mathbf{x}) = f(g^{-1} \cdot \mathbf{x})$$

Where we write the action of g^{-1} on x as $g^{-1} \cdot \mathbf{x}$. This is where the regular group convolution gets its name; because of its use of the regular representation to transform the kernels k used throughout the network.

1.2 Implementing a group in python

Let's start out with a baseclass in which we outline which functions we are going to need when working with groups in our setting. As we're going to use `torch` in our implementation of group convolutional neural networks, let us implement the group as a `torch` module as well.

We first specify a base class `GroupBase` in which we specify all necessary properties and operations that we need in our treatment of group convolutional neural networks. The idea is that in our implementation of group convolutions, implementing these functions is necessary and sufficient for extending group convolutional neural networks to other groups. In other words; if you'd like to implement group convolutions equivariant to a new group you find interesting, just inherit this baseclass, implement its methods, and you're good to go. (In practice this only faithfully works for discrete, compact groups).

```
[ ]: class GroupBase(torch.nn.Module):

         def __init__(self, dimension, identity):
             """ Implements a group.

             @param dimension: Dimensionality of the group (number of dimensions in the basis␣
     ↪of the algebra).
             @param identity: Identity element of the group.
             """
             super().__init__()
             self.dimension = dimension
             self.register_buffer('identity', torch.Tensor(identity))

         def elements(self):
             """ Obtain a tensor containing all group elements in this group.

             """
             raise NotImplementedError()
```

(continues on next page)

(continued from previous page)

```
    def product(self, h, h_prime):
        """ Defines group product on two group elements.

        @param g1: Group element 1
        @param g2: Group element 2
        """
        raise NotImplementedError()

    def inverse(self, h):
        """ Defines inverse for group element.

        @param g: A group element.
        """
        raise NotImplementedError()

    def left_action_on_R2(self, h_batch, x_batch):
        """ Group action of an element from the subgroup H on a vector in R2. For␣
↪efficiency we
        implement this batchwise.

        @param h_batch: Group elements from H.
        @param x_batch: Vectors in R2.
        """
        raise NotImplementedError()

    def left_action_on_H(self, h_batch, h_prime_batch):
        """ Group action of elements of H on other elements in H itself. Comes down to␣
↪group product.
        For efficiency we implement this batchwise. Each element in h_batch is applied␣
↪to each element
        in h_prime_batch.

        @param h_batch: Group elements from H.
        @param h_prime_batch: Other group elements in H.
        """
        raise NotImplementedError()

    def matrix_representation(self, h):
        """ Obtain a matrix representation in R^2 for an element h.

        @param h: Group element
        """
        raise NotImplementedError()

    def determinant(self, h):
        """ Calculate the determinant of the representation of a group element
        h.

        @param g:
        """
        raise NotImplementedError()
```

(continues on next page)

(continued from previous page)

```
    def normalize_group_elements(self, h):
        """ Map the group elements to an interval [-1, 1]. We use this to create
        a standardized input for obtaining weights over the group.

        @param g:
        """
        raise NotImplementedError()
```

1.3 Implementing the cyclic group C_4

As an example, let's discuss a relatively simple group; the group of all 90 rotations of the plane, otherwise known as the cyclic group C_4. Note:

- The set of group elements of C_4 is given by $G := \{e, g, g^2, g^3\}$. We can parameterise these group elements using rotation angles θ, i.e. $e = 0, g = \frac{1}{2}\pi, g^2 = \pi, \ldots$
- The group product is then given by $g \cdot g' := \theta + \theta' \mod 2\pi$.
- The inverse is given by: $g^{-1} = -\theta \mod 2\pi$.
- The group C_4 has an action on the euclidean plane in 2 dimensions $\mathbb{R}^2$ given by a rotation matrix;

$$R_\theta : \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

  This gives us the regular representation $\mathcal{L}_\theta$ on functions f defined over $\mathbb{R}^2$:

$$\mathcal{L}_\theta f(\mathbf{x}) = f(R_{-\theta \mod 2\pi}\mathbf{x}).$$

Let's implement this group!

[ ]:
```
class CyclicGroup(GroupBase):

    def __init__(self, order):
        super().__init__(
            dimension=1,
            identity=[0.]
        )

        assert order > 1
        self.order = torch.tensor(order)

    def elements(self):
        """ Obtain a tensor containing all group elements in this group.

        """
        return torch.linspace(
            start=0,
            end=2 * np.pi * float(self.order - 1) / float(self.order),
            steps=self.order,
            device=self.identity.device
        )
```

(continues on next page)

(continued from previous page)

```
    def product(self, h1, h2):
        """ Defines group product on two group elements of the cyclic group C4.

        @param h1: Group element 1
        @param h2: Group element 2
        """

        # As we directly parameterize the group by its rotation angles, this
        # will be a simple addition. Don't forget the closure property though!

        ## YOUR CODE STARTS HERE ##
        product = ...
        ## AND ENDS HERE ##

        return product

    def inverse(self, h):
        """ Defines group inverse for an element of the cyclic group C4.

        @param h: Group element
        """

        # Implement the inverse operation. Keep the closure property in mind!

        ## YOUR CODE STARTS HERE ##
        inverse = ...
        ## AND ENDS HERE ##

        return inverse

    def left_action_on_R2(self, batch_h, batch_x):
        """ Group action of an element g on a set of vectors in R2.

        @param batch_h: Tensor of group elements. [num_elements]
        @param batch_x: Tensor of vectors in R2. [2, spatial_x, spatial_y]
        """
        # Create a tensor containing representations of each of the group
        # elements in the input. Creates a tensor of size [batch_size, 2, 2].

        ## YOUR CODE STARTS HERE ##
        batched_rep = ...
        ## AND ENDS HERE ##

        # Transform the r2 input grid with each representation to end up with
        # a transformed grid of dimensionality [num_group_elements, spatial_x,
        # spatial_y, 2]. Note the order of the dimensions!

        # Recall that we are working with a left-regular representation,
        # meaning we transform vectors in R^2 through left-matrix multiplication.

        ## YOUR CODE STARTS HERE ##
        out = torch.einsum(...)
```

(continues on next page)

(continued from previous page)

```
        ## AND ENDS HERE ##

        # Afterwards (because grid_sample assummes our grid is y,x instead of x,y)
        # we swap x and y coordinate values with a roll along final dimension.
        return out.roll(shifts=1, dims=-1)

    def left_action_on_H(self, batch_h, batch_h_prime):
        """ Group action of an element h on a set of group elements in H.
        Nothing more than a batchwise group product.

        @param batch_h: Tensor of group elements.
        @param batch_h_prime: Tensor of group elements to apply group product to.
        """
        # The elements in batch_h work on the elements in batch_h_prime directly,
        # through the group product. Each element in batch_h is applied to each element
        # in batch_h_prime.
        transformed_batch_h = self.product(batch_h.repeat(batch_h_prime.shape[0], 1),
                                           batch_h_prime.unsqueeze(-1))
        return transformed_batch_h

    def matrix_representation(self, h):
        """ Obtain a matrix representation in R^2 for an element h.

        @param h: A group element.
        """
        ## YOUR CODE STARTS HERE ##
        representation = ...
        ## AND ENDS HERE ##

        return representation.to(self.identity.device)

    def normalize_group_elements(self, h):
        """ Normalize values of group elements to range between -1 and 1.
        The group elements range from 0 to 2pi * (self.order - 1) / self.order,
        so we normalize by

        @param h: A group element.
        """
        largest_elem = 2 * np.pi * (self.order - 1) / self.order

        return (2*h / largest_elem) - 1.
```

[ ]:
```
# Some tests to verify our implementation.
c4 = CyclicGroup(order=4)
e, g1, g2, g3 = c4.elements()

assert c4.product(e, g1) == g1 and c4.product(g1, g2) == g3
assert c4.product(g1, c4.inverse(g1)) == e

assert torch.allclose(c4.matrix_representation(e), torch.eye(2))
assert torch.allclose(c4.matrix_representation(g2), torch.tensor([[-1, 0], [0, -1]]).
↪float(), atol=1e-6)
```

(continues on next page)

(continued from previous page)

```
assert torch.allclose(c4.left_action_on_R2([g1], torch.tensor([[[0.]], [[1.]]])), torch.
↪tensor([[[0., -1.]]]), atol=1e-7)
```

1.4 Visualizing the group action

Let's play around with the group implementation we have just created! To obtain pixel values for the transformed image, we use pytorch's `grid_sample` function (see documentation).

```
[ ]: img = Image.open(os.path.join(DATASET_PATH, "paprika.tiff"))

     img_tensor = transforms.ToTensor()(img)

     img
```

```
[ ]: # This creates a grid of the pixel locations in our image
     img_grid_R2 = torch.stack(torch.meshgrid(
         torch.linspace(-1, 1, img_tensor.shape[-1]),
         torch.linspace(-1, 1, img_tensor.shape[-2]),
     ))

     # [2, 512, 512] since our image is 2 dimensional and has a width and height of
     # 512 pixels
     img_grid_R2.shape
```

```
[ ]: # let's create the group of 90 degree clockwise rotations
     c4 = CyclicGroup(order=4)
     e, g1, g2, _ = c4.elements()
```

```
[ ]: # Create a counterclockwise rotation of 270 degrees using only e, g1 and g2.

     ## YOUR CODE STARTS HERE ##
     g3 = ...
     ## AND ENDS HERE ##

     assert g3 == c4.elements()[-1]

     # Transform the image grid we just created with the matrix representation of
     # this group element. Note that we implemented this batchwise, so we add a dim.
     transformed_grid = c4.left_action_on_R2(c4.inverse(g3).unsqueeze(0), img_grid_R2)
```

As we'll be using it extensively throughout this tutorial, let's take a closer look at the `grid_sample`. From the pytorch docs:

> Currently, only spatial (4-D) and volumetric (5-D) input are supported.
>
> In the spatial (4-D) case, for input with shape (N, C, H_{in}, W_{in}) and grid with shape $(N, H_{\text{out}}, W_{\text{out}}, 2)$, the output will have shape $(N, C, H_{\text{out}}, W_{\text{out}})$.
>
> Parameters:
>
> input (Tensor) – input of shape $(N, C, H_{\text{in}}, W_{\text{in}})$ (4-D case) or $(N, C, D_{\text{in}}, H_{\text{in}}, W_{\text{in}})$ (5-D case).

For now, we're working in the 4-D setting. We're going to transform a single image with a single transformation, so in our case $N = 1$, the image has 3 channels so $C = 3$, and the height and width we just saw were both $H_{\text{in}} = W_{\text{in}} = 512$.

> grid (Tensor) – flow-field of shape $(N, H_{\text{out}}, W_{\text{out}}, 2)$ (4-D case) or $(N, D_{\text{out}}, H_{\text{out}}, W_{\text{out}}, 3)$ (5-D case)

Remember, this shape is what we matched `transformed_grid` in `C4.left_action_on_R2` to. Currently, we thus expect the transformed grid to be of shape $(1, 512, 512, 2)$.

```
[ ]: transformed_grid.shape
```

```
[ ]: # Let's set up the sampling method with some fixed parameters that we'll use throughout.
     grid_sample = partial(
         torch.nn.functional.grid_sample,
         padding_mode='zeros',
         align_corners=True,
         mode="bilinear"
     )
```

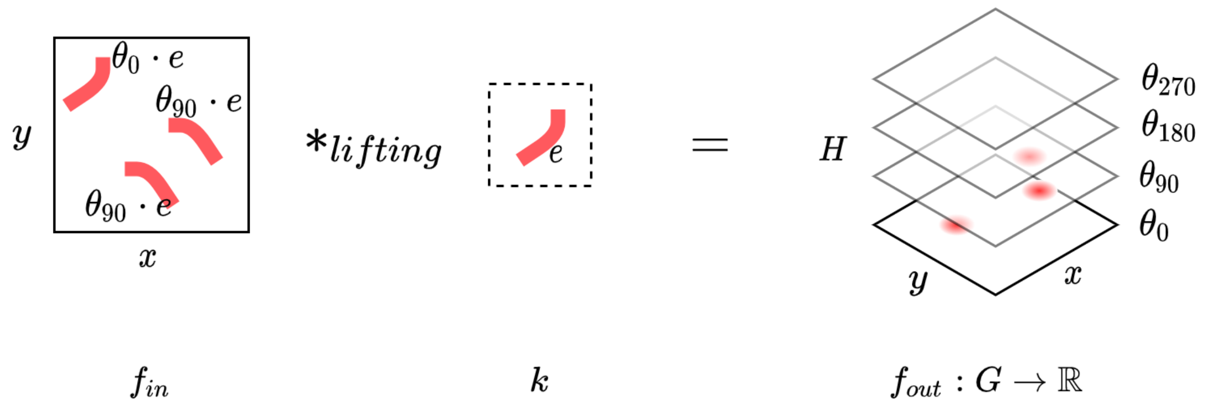
```
[ ]: # This function samples an input tensor based on a grid using interpolation.
     # It is implemented for batchwise operations so we add a dimension to our and input␣
     ↪image.
     transformed_img = grid_sample(img_tensor.unsqueeze(0), transformed_grid)

     # if we turn this back into a PIL image we can see the result of our transformation!
     transforms.ToPILImage()(transformed_img[0])
```

3.5.3 Part 2: Group Equivariant Convolutional Networks

As discussed in the lecture, regular group convolutional neural networks consist of three main elements. The lifting convolution, group convolution and projection operation. We treat these in order.

2.1 Lifting convolution

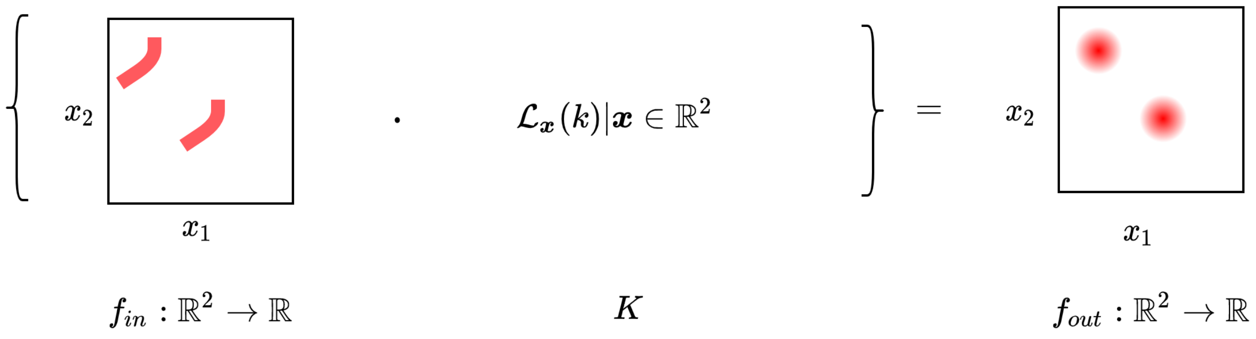
First is the lifting convolution, which disentangles features at any spatial location in the input f_{in} under transformations of H. You may think of this as registering at all locations, for a given feature e, the occurrences of transformed versions of this feature $\mathcal{L}_h(e)$, for $h \in H$. (Instead of $\mathcal{L}_h(e)$, we sometimes write $h \cdot e$ to denote the action of h on e.) The lifting convolution thus maps from $\mathbb{R}^2$ to $G = \mathbb{R}^2 \rtimes H$. As a result, our lifted feature map f_{out} has, besides the usual spatial dimensions, one or more additional group dimensions (dependent on the dimensionality of H).



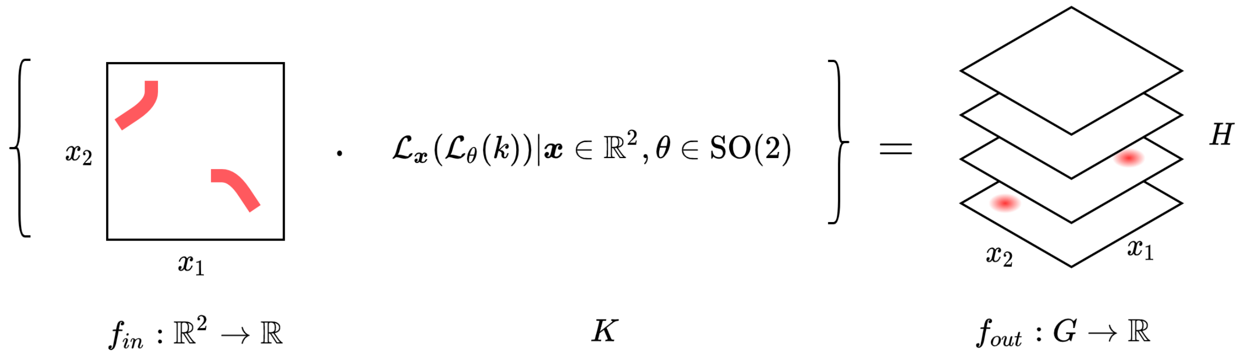
For example, take the group of 90 deg rotations as H, and let's say e is a squiggle of sorts, of which we have three occurences in our input feature map f_{in}. Two are under a 90 deg rotation; $\theta_{90} \cdot e$, and one is under its canonical orientation; $\theta_0 \cdot e$. A lifting convolution with a kernel k which exactly matches the feature e would land responses at different offsets along the group dimension of the feature map; namely one at the spatial feature map corresponding to the group elements θ_0 and two at the spatial feature map corresponding to θ_{90}. See the above figure for an intuition.

2.1.1 Overview

How do we get our convolution operation to pick up features under different transformations of H? Intuitively, it is not that different from the convolution operation as we are familiar with in CNNs. There, we enable the extraction of features at any location by sharing the same kernel over all spatial positions.


x_2
x_1
$\mathcal{L}_{\boldsymbol{x}}(k) | \boldsymbol{x} \in \mathbb{R}^2$
$=$
x_2
x_1
$f_{in} : \mathbb{R}^2 \rightarrow \mathbb{R}$
K
$f_{out} : \mathbb{R}^2 \rightarrow \mathbb{R}$

From our current group theoretic perspective, we interpret this as applying all possible translations $\boldsymbol{x} \in \mathbb{R}^2$ to our kernel k and recording the response we get when we take the inner product of the input f_{in} with this transformed kernel $\mathcal{L}_{\mathbf{x}}(k)$. This operation starts with a feature map defined on $\mathbb{R}^2$ and also yields a feature map defined over $\mathbb{R}^2$. See above.


x_2
x_1
$\mathcal{L}_{\boldsymbol{x}}(\mathcal{L}_{\theta}(k)) | \boldsymbol{x} \in \mathbb{R}^2, \theta \in \mathrm{SO}(2)$
$=$
H
x_2
x_1
$f_{in} : \mathbb{R}^2 \rightarrow \mathbb{R}$
K
$f_{out} : G \rightarrow \mathbb{R}$

Now that we additionally want to register features under different group actions $\mathcal{L}_h$ for $h \in H$, we can do so by simply *also* transforming k with all of their group actions and recording the results. For example, in case of the rotation group C_4, we not only translate, but additionally rotate the kernel k by all possible 90 deg rotations, and record the responses for the resulting transformed kernels!

2.1.2 Implementing the lifting convolution kernel

Let's get to programming. First, we need to define a kernel k which we can transform under arbitrary group actions with $\mathcal{L}_h$. When working with images, a convolution kernel is generally defined as a set of independently sampled weights W defined over an equidistant discretisation of $\mathbb{R}^2$ (pixels are evenly spaced).

Recall that we can express the group action of a group H on functions (such as kernels k) defined over $\mathbb{R}^2$ through the regular representation $\mathcal{L}_h$. The regular representation transforms the function k through a transformation of the domain of the function k. In other words, the regular representation transforms the grid over which the kernel k is defined, to obtain the values for the transformed function $\mathcal{L}_h(k)$.

As such, to define a kernel k which we can transform with the regular representation of a group H, we need to construct a grid over which the kernel values are defined. We can then transform this grid by the action of each group element $h \in H$ to obtain a set of grids corresponding to transformed kernels for each of the group elements of H. Let's get to work!

Notes:

- In implementing the actual lifting and group convolution operations, we will make use of PyTorch's `Conv2D` class. This simplifies our life a lot, since `Conv2D` takes cares of translating the kernels k over all input locations. Hence we do not need to implement the action of the translation group ourselves ($\mathcal{L}_{\mathbf{x}}$), but will still remain translation equivariant! Making our operations compatible with `Conv2D` requires a small amount of trickery, but we will get to that later.

[ ]:
```
class LiftingKernelBase(torch.nn.Module):

    def __init__(self, group, kernel_size, in_channels, out_channels):
        """ Implements a base class for the lifting kernel. Stores the R^2 grid
        over which the lifting kernel is defined and it's transformed copies
        under the action of a group H.

        """
        super().__init__()
        self.group = group

        self.kernel_size = kernel_size
        self.in_channels = in_channels
        self.out_channels = out_channels

        # Create spatial kernel grid. These are the coordinates on which our
        # kernel weights are defined.
        self.register_buffer("grid_R2", torch.stack(torch.meshgrid(
            torch.linspace(-1, 1, self.kernel_size),
            torch.linspace(-1, 1, self.kernel_size),
        )).to(self.group.identity.device))

        # Transform the grid by the elements in this group.
        self.register_buffer("transformed_grid_R2", self.create_transformed_grid_R2())

    def create_transformed_grid_R2(self):
        """Transform the created grid by the group action of each group element.
        This yields a grid (over H) of spatial grids (over R2). In other words,
        a list of grids, each index of which is the original spatial grid transformed by
        a corresponding group element in H.

```

(continues on next page)

(continued from previous page)

```
        """
        # Obtain all group elements.

        ## YOUR CODE STARTS HERE ##
        group_elements = ...
        ## AND ENDS HERE ##

        # Transform the grid defined over R2 with the sampled group elements.
        # Recall how the left-regular representation acts on the domain of a
        # function on R2! (Hint: look closely at the equation given under 1.3)

        ## YOUR CODE STARTS HERE ##
        transformed_grid = ...
        ## AND ENDS HERE ##

        return transformed_grid


    def sample(self, sampled_group_elements):
        """ Sample convolution kernels for a given number of group elements

        arguments should include:
        :param sampled_group_elements: the group elements over which to sample
            the convolution kernels

        should return:
        :return kernels: filter bank extending over all input channels,
            containing kernels transformed for all output group elements.
        """
        raise NotImplementedError()
```

[ ]:
```
# Let's check whether our implementation works correctly. First we inspect the
# shape of our transformed grids to assess whether this is correct.
lifting_kernel_base = LiftingKernelBase(
    group=CyclicGroup(order=4),
    kernel_size=2,
    in_channels=1,
    out_channels=1
)

assert lifting_kernel_base.transformed_grid_R2.shape == torch.Size([4, 2, 2, 2])
```

[ ]:
```
# Let's visualize the transformed kernel grids!
lifting_kernel_base = LiftingKernelBase(
    group=CyclicGroup(order=4),
    kernel_size=7,
    in_channels=1,
    out_channels=1
)
```

(continues on next page)

(continued from previous page)

```
transformed_grid_R2 = lifting_kernel_base.transformed_grid_R2

# The grid has a shape of [num_group_elements, kernel_size, kernel_size, dim_fmap_
↪domain(R^2)]
transformed_grid_R2.shape
```

[ ]:
```
plt.rcParams['figure.figsize'] = [12, 3]

# Create [group_elements] figures
fig, ax = plt.subplots(1, transformed_grid_R2.shape[0])

# Fold both spatial dimensions into a single dimension
transformed_grid_R2 = transformed_grid_R2.reshape(transformed_grid_R2.shape[0],
                                                  transformed_grid_R2.
↪shape[1]*transformed_grid_R2.shape[2],
                                                  2).numpy()

# Visualize the transformed kernel grids. We mark the same cornerpoint by a blue 'x' in␣
↪all grids as reference point.
for group_elem in range(transformed_grid_R2.shape[0]):
    ax[group_elem].scatter(transformed_grid_R2[group_elem, 0, 0], transformed_grid_
↪R2[group_elem, 0, 1], marker='x', c='b')
    ax[group_elem].scatter(transformed_grid_R2[group_elem, 1:, 0], transformed_grid_
↪R2[group_elem, 1:, 1], c='r')

fig.text(0.5, 0.04, 'Group elements', ha='center')
plt.show()
```

If your code is correctly implemented, you should see the counter-clockwise rotation action happening!

At this point we have a set of grids transformed under the operation of the group H. We now need to decide how we are going to sample kernel values at the grid points in each of these grids. This is where the first major hurdle in applying GCNNs occurs.

Whereas conventional CNNs get away with sharing the same set of weights over all spatial positions (which is due to the fact that we translate the kernel only with steps that match whole pixel-distances), for arbitrary groups H we may require kernel values for grid points that lie off the pixel grid of the kernel under its canonical transformation.

Of course, we could (and in fact will) use interpolation to obtain kernel values for grid locations between pixel locations, but this may be limiting in expressivity and will introduce interpolation artefacts!

Notes:

- When we are implementing equivariance for the group of 90 deg rotations, $H = C_4$, we of course could get away without using interpolation, since all transformed grids lie share the same locations. We could implement the group action of this particular group through a permutation of the weights. But we'd like to be more general than that in this tutorial, so we'll go with interpolation instead! Luckily PyTorch has a function that allows us to sample an input on a grid; we will use PyTorch's `grid_sample` function for interpolation!

[ ]:
```
class InterpolativeLiftingKernel(LiftingKernelBase):

    def __init__(self, group, kernel_size, in_channels, out_channels):
        super().__init__(group, kernel_size, in_channels, out_channels)

```

(continues on next page)

(continued from previous page)

```
        # Create and initialise a set of weights, we will interpolate these
        # to create our transformed spatial kernels.
        self.weight = torch.nn.Parameter(torch.zeros((
            self.out_channels,
            self.in_channels,
            self.kernel_size,
            self.kernel_size
        ), device=self.group.identity.device))

        # Initialize weights using kaiming uniform intialisation.
        torch.nn.init.kaiming_uniform_(self.weight.data, a=math.sqrt(5))

    def sample(self):
        """ Sample convolution kernels for a given number of group elements

        should return:
        :return kernels: filter bank extending over all input channels,
            containing kernels transformed for all output group elements.
        """
        # First, we fold the output channel dim into the input channel dim;
        # this allows us to transform the entire filter bank in one go using the
        # torch grid_sample function.

        # Next, we want a transformed set of weights for each group element so
        # we repeat the set of spatial weights along the output group axis.

        ## YOUR CODE STARTS HERE ##
        weight = ...
        ## AND ENDS HERE ##

        # Check whether the weight has the expected shape.
        assert weight.shape == torch.Size((
            self.group.elements().numel(),
            self.out_channels * self.in_channels,
            self.kernel_size,
            self.kernel_size
        ))

        # Sample the transformed kernels.
        transformed_weight = grid_sample(
            weight,
            self.transformed_grid_R2
        )

        # Separate input and output channels.
        transformed_weight = transformed_weight.view(
            self.group.elements().numel(),
            self.out_channels,
            self.in_channels,
            self.kernel_size,
            self.kernel_size
        )
```

(continues on next page)

(continued from previous page)

```
        # Put the output channel dimension before the output group dimension.
        transformed_weight = transformed_weight.transpose(0, 1)

        return transformed_weight
```

[ ]:
```
ik = InterpolativeLiftingKernel(
    group=CyclicGroup(order=4),
    kernel_size=5,
    in_channels=2,
    out_channels=1
)

weights = ik.sample()
```

Let's visualize the weights we sampled from our lifting convolution kernel!

[ ]:
```
plt.rcParams['figure.figsize'] = [10, 5]

# Pick an output channel to visualize
out_channel_idx = 0

# Create [in_channels, group_elements] figures
fig, ax = plt.subplots(weights.shape[2], weights.shape[1])

for in_channel in range(weights.shape[2]):
    for group_elem in range(weights.shape[1]):
        ax[in_channel, group_elem].imshow(
            weights[out_channel_idx, group_elem, in_channel, :, :].detach().numpy()
        )

fig.text(0.5, 0.04, 'Group elements', ha='center')
fig.text(0.04, 0.5, 'Input channels', va='center', rotation='vertical')

plt.show()
```

As we can see, spatial kernel rotates under the action of the rotation group elements!

2.1.3 Implementing the lifting convolution

Finally, we can implement the lifting convolution operation! This class should take as input a feature map defined over $\mathbb{R}^2$, and spit out a feature map over $\mathbb{R}^2 \rtimes H$, where features under different transformations $h \in H$ are disentangled along the H axis!

Notes:

- To prevent having to implement our own implementation of the convolution operation, and to leverage the highly optimized pytorch `Conv2D` class, we use some neat tricks in our lifting (and group) convolution classes. Normally, a convolution layer applies a set of n spatial kernels throughout the input, where n is the number of output channels of the convolution operation. Because we now also have `num_group_elem` transformed versions of these kernels, which we want to apply everywhere in the input, we can trick PyTorch by having it treat every transformation of the same spatial kernel as a separate output channel. To do this, we simply reshape our set of `[out_channels, num_group_elem, in_channels, kernel_size, kernel_size]` kernels into a set of `[out_channels x num_group_elem, in_channels, kernel_size, kernel_size]` kernels. See below!

- As mentioned, a great additional benefit of using the PyTorch `Conv2D` class is that we are not required to obtain translated kernels $\mathcal{L}_{\mathbf{x}}(k)$ ourselves, PyTorch takes care of that!

[ ]:
```
class LiftingConvolution(torch.nn.Module):

    def __init__(self, group, in_channels, out_channels, kernel_size):
        super().__init__()

        self.kernel = InterpolativeLiftingKernel(
            group=group,
            kernel_size=kernel_size,
            in_channels=in_channels,
            out_channels=out_channels
        )

    def forward(self, x):
        """ Perform lifting convolution

        @param x: Input sample [batch_dim, in_channels, spatial_dim_1,
            spatial_dim_2]
        @return: Function on a homogeneous space of the group
            [batch_dim, out_channels, num_group_elements, spatial_dim_1,
            spatial_dim_2]
        """

        # Obtain convolution kernels transformed under the group.

        ## YOUR CODE STARTS HERE ##
        conv_kernels = ...
        ## AND ENDS HERE ##

        # Apply lifting convolution. Note that using a reshape we can fold the
        # group dimension of the kernel into the output channel dimension. We
        # treat every transformed kernel as an additional output channel. This
        # way we can use pytorch's conv2d function!

        # Question: Do you see why we (can) do this?

        ## YOUR CODE STARTS HERE ##
        x = torch.nn.functional.conv2d(
            input=x,
            weight=conv_kernels.reshape(
                ...
            ),
        )
        ## AND ENDS HERE ##

        # Reshape [batch_dim, in_channels * num_group_elements, spatial_dim_1,
        # spatial_dim_2] into [batch_dim, in_channels, num_group_elements,
        # spatial_dim_1, spatial_dim_2], separating channel and group
        # dimensions.
```

(continues on next page)

(continued from previous page)

```
        x = x.view(
            -1,
            self.kernel.out_channels,
            self.kernel.group.elements().numel(),
            x.shape[-1],
            x.shape[-2]
        )

        return x
```

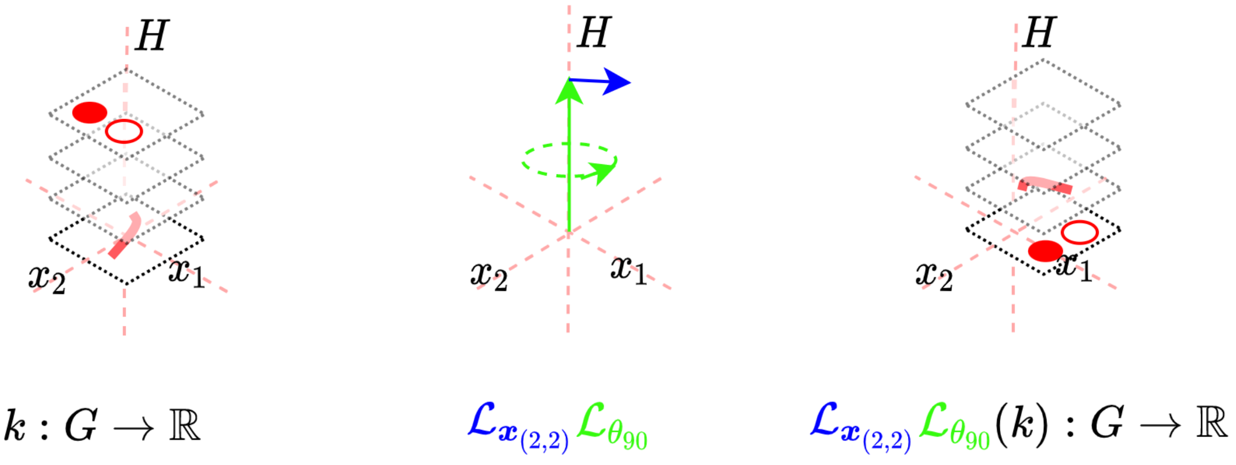
```
[ ]: lifting_conv = LiftingConvolution(
         group=CyclicGroup(order=4),
         kernel_size=5,
         in_channels=3,
         out_channels=8
     )
```

2.2 Group convolution

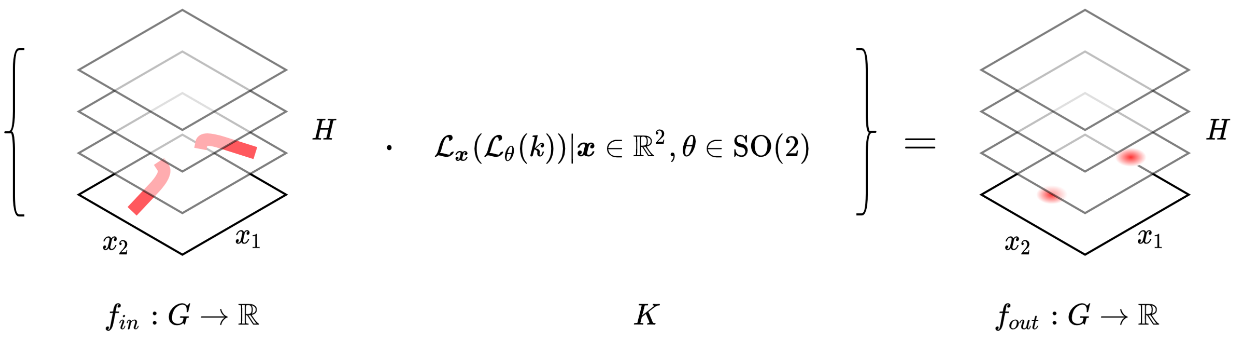
Now that we have a way to obtain feature maps defined over the group G from input functions defined over $\mathbb{R}^2$, let's move on to implementing a convolutional layer that fully operates on the group $G : \mathbb{R}^2 \rtimes H$. Note that the input feature map at this stage, f_{in}, has, besides the usual spatial dimensions defined over $\mathbb{R}^2$, one or more additional group dimensions defined over H. As such, the group convolution operation, $*_{group}$, maps from a function on the group f_{in} to another function on the group f_{out}.

2.2.1 Overview

Since the input to the group convolution layer now contains additional group dimensions; $f_{in} : \mathbb{R}^2 \rtimes H \to \mathbb{R}$, we need to convolve it with a kernel k_{group} that is also defined over the entire group; $k_{group} : \mathbb{R}^2 \rtimes H \to \mathbb{R}$. This is in contrast to the lifting convolution, where $k_{lifting}$ was only defined over the spatial domain $\mathbb{R}^2$. You could think of this kernel k_{group} as a stack of spatial kernels, a separate one for each group element $h \in H$. Importantly, since k is now *also* defined on a grid over H, the group H now also has an action *on* this H-axis of k. For example, in our case of $H = C_4$, elements $\theta \in C_4$ now not only have a rotating action on the spatial domain of the kernel, but also a translating action along the group axis. Hence, applying a group element $\theta \in C_4$ leads to a twist-shift of the k along the group axis. See below! (Remember $H = C_4$ is periodic)


H
x_2
x_1
$k : G \to \mathbb{R}$
$\mathcal{L}_{\boldsymbol{x}_{(2,2)}} \mathcal{L}_{\theta_{90}}$
$\mathcal{L}_{\boldsymbol{x}_{(2,2)}} \mathcal{L}_{\theta_{90}}(k) : G \to \mathbb{R}$

Apart from this difference, the group convolution operator works in much the same way as the liting operator. We again transform the kernel k_{group} with the actions of the group H and $\mathbb{R}^2$ and obtain responses for the inner product of this kernel with the input (although again, we let PyTorch do all the work for the translation group). See below for an intuition.


H
x_2
x_1
$\cdot$
$\mathcal{L}_{\boldsymbol{x}}(\mathcal{L}_\theta(k)) | \boldsymbol{x} \in \mathbb{R}^2, \theta \in \mathrm{SO}(2)$
$=$
$f_{in} : G \to \mathbb{R}$
K
$f_{out} : G \to \mathbb{R}$

2.2.2 Implementing the group convolution kernel

Again, let's define a kernel k which we can transform with the group action. Now, our kernel grid is not only defined over $\mathbb{R}^2$, but additionally over H.

Notes:

- Since the grid over H is made up of elements $h' \in H$, transforming the grid over H with (another) group element $h \in H$ comes out to applying the group product of h with each grid element h'.
- Because we are working with semidirect product groups $\mathbb{R}^2 \rtimes H$, we can transform the $\mathbb{R}^2$ and H dimensions of our grids separately before combining them into a shared grid over $\mathbb{R}^2 \rtimes H$!

[ ]:
```
class GroupKernelBase(torch.nn.Module):

    def __init__(self, group, kernel_size, in_channels, out_channels):
        """ Implements base class for the group convolution kernel. Stores grid
        defined over the group R^2 \rtimes H and it's transformed copies under
        all elements of the group H.

```

(continues on next page)

(continued from previous page)

```
        """
        super().__init__()
        self.group = group

        self.kernel_size = kernel_size
        self.in_channels = in_channels
        self.out_channels = out_channels

        # Create a spatial kernel grid
        self.register_buffer("grid_R2", torch.stack(torch.meshgrid(
            torch.linspace(-1, 1, self.kernel_size),
            torch.linspace(-1, 1, self.kernel_size),
        )).to(self.group.identity.device))

        # The kernel grid now also extends over the group H, as our input
        # feature maps contain an additional group dimension
        self.register_buffer("grid_H", self.group.elements())

        self.register_buffer("transformed_grid_R2xH", self.create_transformed_grid_
↪R2xH())

    def create_transformed_grid_R2xH(self):
        """Transform the created grid over R^2 \rtimes H by the group action of
        each group element in H.

        This yields a set of grids over the group. In other words, a list of
        grids, each index of which is the original grid over G transformed by
        a corresponding group element in H.
        """
        # Sample the group H.

        ## YOUR CODE STARTS HERE ##
        group_elements = ...
        ## AND ENDS HERE ##

        # Transform the grid defined over R2 with the sampled group elements.

        ## YOUR CODE STARTS HERE ##
        transformed_grid_R2 = ...
        ## AND ENDS HERE ##

        # Transform the grid defined over H with the sampled group elements.

        ## YOUR CODE STARTS HERE ##
        transformed_grid_H = ...
        ## AND ENDS HERE ##

        # Rescale values to between -1 and 1, we do this to please the torch
        # grid_sample function.
        transformed_grid_H = self.group.normalize_group_elements(transformed_grid_H)

        # Create a combined grid as the product of the grids over R2 and H
```

(continues on next page)

(continued from previous page)

```
        # repeat R2 along the group dimension, and repeat H along the spatial dimension
        # to create a [output_group_elem, num_group_elements, kernel_size, kernel_size,␣
↪3] grid
        transformed_grid = torch.cat(
            (
                transformed_grid_R2.view(
                    group_elements.numel(),
                    1,
                    self.kernel_size,
                    self.kernel_size,
                    2,
                ).repeat(1, group_elements.numel(), 1, 1, 1),
                transformed_grid_H.view(
                    group_elements.numel(),
                    group_elements.numel(),
                    1,
                    1,
                    1,
                ).repeat(1, 1, self.kernel_size, self.kernel_size, 1, )
            ),
            dim=-1
        )
        return transformed_grid


    def sample(self, sampled_group_elements):
        """ Sample convolution kernels for a given number of group elements

        arguments should include:
        :param sampled_group_elements: the group elements over which to sample
            the convolution kernels

        should return:
        :return kernels: filter bank extending over all input channels,
            containing kernels transformed for all output group elements.
        """
        raise NotImplementedError()
```

Let's get some intuition for what is happening with our grid when we apply the group action of H to it. First we will inspect the action on $\mathbb{R}^2$.

```
[ ]: group_kernel_base = GroupKernelBase(
         group=CyclicGroup(order=4),
         kernel_size=7,
         in_channels=1,
         out_channels=1
     )

     # Sample the group.
     group_elements = group_kernel_base.group.elements()

     # Transform the grid defined over R2 with the sampled group elements.
```

(continues on next page)

(continued from previous page)

```
transformed_grid_R2 = group_kernel_base.group.left_action_on_R2(
    group_kernel_base.group.inverse(group_elements),
    group_kernel_base.grid_R2
)
```

```
[ ]: plt.rcParams['figure.figsize'] = [10, 3]

# Create [group_elements] figures.
fig, ax = plt.subplots(1, transformed_grid_R2.shape[0])

# Fold both spatial dimensions into a single dimension.
transformed_grid_R2 = transformed_grid_R2.reshape(transformed_grid_R2.shape[0],
                                                  transformed_grid_R2.
↪shape[1]*transformed_grid_R2.shape[2],
                                                  2).numpy()

# Visualize the transformed kernel grids. We mark the same cornerpoint by a blue 'x' in␣
↪all grids as reference point.
for group_elem in range(transformed_grid_R2.shape[0]):
    ax[group_elem].scatter(transformed_grid_R2[group_elem, 0, 0], transformed_grid_
↪R2[group_elem, 0, 1], marker='x', c='b')
    ax[group_elem].scatter(transformed_grid_R2[group_elem, 1:, 0], transformed_grid_
↪R2[group_elem, 1:, 1], c='r')

fig.text(0.5, 0.04, 'Group elements', ha='center')

plt.show()
```

As we can see, this part of the grid, and what happens to it, is identical to the grid we saw in the lifting convolution. Note however, that this is only the spatial dimensions of the grid over which a group convolution kernel is defined. Let's move on to the grid over H.

```
[ ]: plt.rcParams['figure.figsize'] = [5.5, 3]
# Transform the grid defined over H with the sampled group elements.
transformed_grid_H = group_kernel_base.group.left_action_on_H(
    group_kernel_base.group.inverse(group_elements), group_kernel_base.grid_H
)

# Create [group_elements] figures.
fig, ax = plt.subplots(1, transformed_grid_H.shape[0])

# Visualize the transformed kernel grids. We mark the same cornerpoint by a blue 'x' in␣
↪all grids as reference point.
for group_elem in range(transformed_grid_H.shape[0]):
    ax[group_elem].scatter(torch.zeros_like(transformed_grid_H[group_elem, 0]),␣
↪transformed_grid_H[group_elem, 0], marker='x', c='b')
    ax[group_elem].scatter(torch.zeros_like(transformed_grid_H[group_elem, 1:]),␣
↪transformed_grid_H[group_elem, 1:], c='r')
    ax[group_elem].set_xticks([])

fig.text(0.5, 0.04, 'Group elements', ha='center')
```

(continues on next page)

(continued from previous page)

```
# Remove the xticks, our group is 1D!
plt.show()
```

In our current setting, H is one-dimensional, and as we can see, transforming the grid over H with all group elements of H leads to a translation over the group. Next, let's see what happens when we combine these grids.

```
[ ]: plt.rcParams['figure.figsize'] = [10, 3]
transformed_grid_R2xH = group_kernel_base.transformed_grid_R2xH

# Create [group_elements] figures.
fig, ax = plt.subplots(1, transformed_grid_H.shape[0], subplot_kw=dict(projection='3d'))

# Flatten spatial and group grid dimensions.
transformed_grid_R2xH = transformed_grid_R2xH.reshape(transformed_grid_R2xH.shape[0],
                                                      transformed_grid_R2xH.shape[1] *␣
↪transformed_grid_R2xH.shape[2] * transformed_grid_R2xH.shape[3],
                                                      transformed_grid_R2xH.shape[4])

# Visualize the transformed kernel grids. We mark the same row by a blue 'x' in all grids␣
↪as reference point.
for group_elem in range(transformed_grid_R2xH.shape[0]):
    ax[group_elem].scatter(transformed_grid_R2xH[group_elem, 0:7, 0],
                           transformed_grid_R2xH[group_elem, 0:7, 1],
                           transformed_grid_R2xH[group_elem, 0:7, 2],
                           marker='x',
                           c='b')
    ax[group_elem].scatter(transformed_grid_R2xH[group_elem, 7:, 0],
                           transformed_grid_R2xH[group_elem, 7:, 1],
                           transformed_grid_R2xH[group_elem, 7:, 2],
                           c='r')

fig.text(0.5, 0.04, 'Group elements', ha='center')

plt.show()
```

As we can see, under the application of group elements $h' \in H$ the grid defined over $\mathbb{R}^2 \rtimes H$ rotates over the spatial dimensions, and shifts along the group dimension!

Let's now implement the group kernel using interpolation as well.

Notes:

- Luckily, `grid_sample` also supports 3D inputs, so we can continue using it.
- For multidimensional groups H the following implementation won't work, as this would require kernels defined on grids with dimensionality > 3, which `grid_sample` does not support. To resolve this, one could implement the sampling of the weights along the H dimension using a translation of the weight matrix along the H dimensions, and only interpolate over the spatial dimensions. This is possible because we don't end up between grid points along the group dimension H (remember the closure constraint of the group product?).
- The C_4 group exhibits periodicity along the group axis, so our kernels should too. Although we correctly implemented the group product to reflect this, `grid_sample` doesn't know about the periodicity of the weights in its interpolation. This shouldn't be a problem since, again because of the closure constraint, we should always end up exactly on grid points along the group axis, meaning no interpolation is necessary in that direction. In practice, because of the way `grid_sample` is implemented, we may encouter some small interpolation artefacts

because of this.

```
[ ]: class InterpolativeGroupKernel(GroupKernelBase):

         def __init__(self, group, kernel_size, in_channels, out_channels):
             super().__init__(group, kernel_size, in_channels, out_channels)

             # Create and initialise a set of weights, we will interpolate these
             # to create our transformed spatial kernels. Note that our weight
             # now also extends over the group H.

             ## YOUR CODE STARTS HERE ##
             self.weight = torch.nn.Parameter(torch.zeros((
                 ...
             ), device=self.group.identity.device))
             ## AND ENDS HERE ##

             # initialize weights using kaiming uniform intialisation.
             torch.nn.init.kaiming_uniform_(self.weight.data, a=math.sqrt(5))

         def sample(self):
             """ Sample convolution kernels for a given number of group elements

             should return:
             :return kernels: filter bank extending over all input channels,
                 containing kernels transformed for all output group elements.
             """
             # First, we fold the output channel dim into the input channel dim;
             # this allows us to transform the entire filter bank in one go using the
             # torch grid_sample function.

             # Next, we want a transformed set of weights for each group element so
             # we repeat the set of spatial weights along the output group axis.

             ## YOUR CODE STARTS HERE ##
             weight = ...
             ## AND ENDS HERE ##

             assert weight.shape == torch.Size((
                 self.group.elements().numel(),
                 self.out_channels * self.in_channels,
                 self.group.elements().numel(),
                 self.kernel_size,
                 self.kernel_size
             ))

             # Sample the transformed kernels using the grid_sample function.
             transformed_weight = grid_sample(
                 weight,
                 self.transformed_grid_R2xH,
             )

             # Separate input and output channels. Note we now have a notion of
```

(continues on next page)

(continued from previous page)

```
        # input and output group dimensions in our weight matrix!
        transformed_weight = transformed_weight.view(
            self.group.elements().numel(), # Output group elements (like in the lifting␣
↪convolution)
            self.out_channels,
            self.in_channels,
            self.group.elements().numel(), # Input group elements (due to the additional␣
↪dimension of our feature map)
            self.kernel_size,
            self.kernel_size
        )

        # Put the output channel dimension before the output group dimension.
        transformed_weight = transformed_weight.transpose(0, 1)

        return transformed_weight
```

```
[ ]: ik = InterpolativeGroupKernel(
         group=CyclicGroup(order=4),
         kernel_size=5,
         in_channels=2,
         out_channels=8
     )
```

```
[ ]: weights = ik.sample()
     weights.shape
```

Let's visualize the sampled group convolution kernels! We visualize our 3D kernels in 2D by folding the input group dimension into the first spatial dimension. In doing so, we create a 2D flattened version of the 3D group convolution kernel, where spatial kernels corresponding to the different group elements lie along the spatial dimension. Each channel goes from `[num_group_elem, kernel_size, kernel_size]` to `[num_group_elem x kernel_size, kernel_size]`.

To clearly see what happens to the group convolution kernel under transformation of the group H, we outline the spatial kernel corresponding to the first input group element in red. For subsequent transformations we can see this spatial kernel. See below!

```
[ ]: plt.rcParams['figure.figsize'] = [10, 10]

     # For ease of viewing, we fold the input group dimension into the spatial x dimension
     weights_t = weights.view(
         weights.shape[0],
         weights.shape[1],
         weights.shape[2],
         weights.shape[3] * weights.shape[4],
         weights.shape[5]
     )

     # pick an output channel to visualize
     out_channel_idx = 0

     # create [in_channels, group_elements] figures
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(weights.shape[2], weights.shape[1])

for in_channel in range(weights.shape[2]):
    for group_elem in range(weights.shape[1]):
        ax[in_channel, group_elem].imshow(
            weights_t[out_channel_idx, group_elem, in_channel, :, :].detach()
        )

        # Outline the spatial kernel corresponding to the first group element under␣
↪canonical transformation
        rect = matplotlib.patches.Rectangle(
            (-0.5, group_elem * weights_t.shape[-1] - 0.5), weights_t.shape[-1], weights_
↪t.shape[-1], linewidth=5, edgecolor='r', facecolor='none')
        ax[in_channel, group_elem].add_patch(rect)

fig.text(0.5, 0.04, 'Group elements', ha='center')
fig.text(0.04, 0.5, 'Input channels / input group elements', va='center', rotation=
↪'vertical')

plt.show()
```

We see the same twist shift motion as we saw with the kernel grids!

2.2.3 Implementing the group convolution

The next step is implementing the group convolution operation.

Notes:

- We would still like to use PyTorch's `Conv2D` implementation, but we're now faced with an additional problem; the group dimension in the input feature map. Luckily we can resolve this problem in much the same way; normally a 2D convolution layer integrates over a local neighbourhood of all input channels. We would now additionally like to integrate over the entire group. Thus, we can simply treat the group dimensions in the input feature map as additional channel dimensions! We achieve this by folding our input group dimension into the input channel dimension; f_{in} is reshaped from `[batch, in_channels, num_group_elem, spatial_1, spatial_2]` into `[batch, in_channels x num_group_elem, spatial_1, spatial_2]`.
- To match this, and to apply the same trick as we did in the lifting convolution to get results for each separate group element in the *output*, we also reshape our kernel from `[out_channels, num_group_elem, in_channels, num_group_elem, kernel_size, kernel_size]` to `[out_channels x num_group_elem, in_channels x num_group_elem, kernel_size, kernel_size]`. See below!

[ ]:
```
class GroupConvolution(torch.nn.Module):

    def __init__(self, group, in_channels, out_channels, kernel_size):
        super().__init__()

        self.kernel = InterpolativeGroupKernel(
            group=group,
            kernel_size=kernel_size,
            in_channels=in_channels,
            out_channels=out_channels
        )
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        """ Perform group convolution

        @param x: Input sample [batch_dim, in_channels, group_dim, spatial_dim_1,
            spatial_dim_2]
        @return: Function on a homogeneous space of the group
            [batch_dim, out_channels, num_group_elements, spatial_dim_1,
            spatial_dim_2]
        """

        # We now fold the group dimensions of our input into the input channel
        # dimension.

        ## YOUR CODE STARTS HERE ##
        x = x.reshape(
            ...
        )
        ## AND ENDS HERE ##

        # We obtain convolution kernels transformed under the group.

        ## YOUR CODE STARTS HERE ##
        conv_kernels = ...
        ## AND ENDS HERE ##

        # Apply group convolution, note that the reshape folds the 'output' group
        # dimension of the kernel into the output channel dimension, and the
        # 'input' group dimension into the input channel dimension.

        # Question: Do you see why we (can) do this?

        ## YOUR CODE STARTS HERE ##
        x = ...
        ## AND ENDS HERE ##

        # Reshape [batch_dim, in_channels * num_group_elements, spatial_dim_1,
        # spatial_dim_2] into [batch_dim, in_channels, num_group_elements,
        # spatial_dim_1, spatial_dim_2], separating channel and group
        # dimensions.
        x = x.view(
            -1,
            self.kernel.out_channels,
            self.kernel.group.elements().numel(),
            x.shape[-1],
            x.shape[-2],
        )

        return x
```

2.3 Projection to obtain invariance and tying everything together

Up until now, our feature maps equivary with the group action of $\mathbb{R}^2 \rtimes H$; our feature maps are defined over $\mathbb{R}^2 \rtimes H$. Usually in a CNN, a series of convolutional layers build a representation, which is followed by a (number of) linear layer(s). To create a GCNN using our lifting and group convolution operations that is fully invariant to the action of the group, we must apply a projection operation invariant to the action of the group to our feature map, to reduce its dimensionality from `[batch, channels, num_group_elem, spatial_1, spatial_2]` to `[batch, channels]` or even . The representation that we obtain then is fully invariant to the group. This representation is pushed through a final linear layer to yield a classification.

Below, we build a small GCNN from our implemented PyTorch modules.

Notes:

- We use a mean-pooling operation to pool over group and spatial dimensions, but we could also use max or min pooling, or any other operation invariant to the group.

[ ]:
```
from torch.nn import AdaptiveAvgPool3d


class GroupEquivariantCNN(torch.nn.Module):

    def __init__(self, group, in_channels, out_channels, kernel_size, num_hidden, hidden_
↪channels):
        super().__init__()

        # Create the lifing convolution.

        ## YOUR CODE STARTS HERE ##
        self.lifting_conv = ...
        ## AND ENDS HERE ##

        # Create a set of group convolutions.
        self.gconvs = torch.nn.ModuleList()

        ## YOUR CODE STARTS HERE ##
        for i in range(num_hidden):
            self.gconvs.append(
                ...
            )
        ## AND ENDS HERE ##

        # Create the projection layer. Hint: check the import at the top of
        # this cell.

        ## YOUR CODE STARTS HERE ##
        self.projection_layer = ...
        ## AND ENDS HERE ##

        # And a final linear layer for classification.
        self.final_linear = torch.nn.Linear(hidden_channels, out_channels)

    def forward(self, x):

        # Lift and disentangle features in the input.
```
(continues on next page)

(continued from previous page)

```
        x = self.lifting_conv(x)
        x = torch.nn.functional.layer_norm(x, x.shape[-4:])
        x = torch.nn.functional.relu(x)

        # Apply group convolutions.
        for gconv in self.gconvs:
            x = gconv(x)
            x = torch.nn.functional.layer_norm(x, x.shape[-4:])
            x = torch.nn.functional.relu(x)

        # to ensure equivariance, apply max pooling over group and spatial dims.
        x = self.projection_layer(x).squeeze()

        x = self.final_linear(x)
        return x
```

To compare, let's create a more or less identical CNN. The only difference here is that this network consists of regular convolution operations.

[ ]:
```
class CNN(torch.nn.Module):

    def __init__(self, in_channels, out_channels, kernel_size, num_hidden, hidden_
↪channels):
        super().__init__()

        self.first_conv = torch.nn.Conv2d(
            in_channels=in_channels,
            out_channels=hidden_channels,
            kernel_size=kernel_size
        )

        self.convs = torch.nn.ModuleList()
        for i in range(num_hidden):
            self.convs.append(
                torch.nn.Conv2d(
                    in_channels=hidden_channels,
                    out_channels=hidden_channels,
                    kernel_size=kernel_size
                )
            )

        self.final_linear = torch.nn.Linear(hidden_channels, out_channels)

    def forward(self, x):

        x = self.first_conv(x)
        x = torch.nn.functional.layer_norm(x, x.shape[-3:])
        x = torch.nn.functional.relu(x)

        for conv in self.convs:
            x = conv(x)
            x = torch.nn.functional.layer_norm(x, x.shape[-3:])
```

(continues on next page)

(continued from previous page)

```
            x = torch.nn.functional.relu(x)

        # Apply average pooling over remaining spatial dimensions.
        x = torch.nn.functional.adaptive_avg_pool2d(x, 1).squeeze()

        x = self.final_linear(x)
        return x
```

3.5.4 Part 3: Experimenting with our implementation

Note that for ease of model training and tracking, we additionally make use of `pytorch-lightning` as in the previous practicals.

3.1 Generalization to the group action

To show the generalization capabilities of regular group convolutional networks, we will train this model on the MNIST training dataset, but evaluate it on an augmented version of the MNIST test set in which each image is randomly rotated by a continuous rotation between $[0, 2\pi]$.

[ ]:
```
# We normalize the training data.
train_transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                                  torchvision.transforms.Normalize((0.
↪1307,), (0.3081,))
                                                  ])

# To demonstrate the generalization capabilities our rotation equivariant layers bring,␣
↪we apply a random
# rotation between 0 and 360 deg to the test set.
test_transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                                 torchvision.transforms.RandomRotation(
                                                     [0, 360],
                                                     torchvision.transforms.
↪InterpolationMode.BILINEAR,
                                                     fill=0),
                                                 torchvision.transforms.Normalize((0.
↪1307,), (0.3081,))
                                                 ])

# We demonstrate our models on the MNIST dataset.
train_ds = torchvision.datasets.MNIST(root=DATASET_PATH, train=True, transform=train_
↪transform, download=True)
test_ds = torchvision.datasets.MNIST(root=DATASET_PATH, train=False, transform=test_
↪transform)
train_loader = torch.utils.data.DataLoader(train_ds, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_ds, batch_size=64, shuffle=False)

# Set the random seed for reproducibility.
pl.seed_everything(12)
```

Let’s visualize some of the training and test images. As we can see the test images are randomly rotated.

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

img_grid = torchvision.utils.make_grid(torch.stack(images + orig_images, dim=0), nrow=4,␣
↪normalize=True, pad_value=0.5)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("Images sampled from the MNIST train set, augmented with test transforms.")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

```
[ ]: class DataModule(pl.LightningModule):

    def __init__(self, model_name, model_hparams, optimizer_name, optimizer_hparams):
        """
        Inputs:
            model_name - Name of the model/CNN to run. Used for creating the model (see␣
↪function below)
            model_hparams - Hyperparameters for the model, as dictionary.
            optimizer_name - Name of the optimizer to use. Currently supported: Adam, SGD
            optimizer_hparams - Hyperparameters for the optimizer, as dictionary. This␣
↪includes learning rate, weight decay, etc.
        """
        super().__init__()
        # Exports the hyperparameters to a YAML file, and create "self.hparams" namespace
        self.save_hyperparameters()
        # Create model
        self.model = create_model(model_name, model_hparams)
        # Create loss module
        self.loss_module = nn.CrossEntropyLoss()

    def forward(self, imgs):
        return self.model(imgs)

    def configure_optimizers(self):
        # AdamW is Adam with a correct implementation of weight decay (see here for␣
↪details: https://arxiv.org/pdf/1711.05101.pdf)
        optimizer = optim.AdamW(
            self.parameters(), **self.hparams.optimizer_hparams)
        return [optimizer], []

    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()
```

(continues on next page)

(continued from previous page)

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

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

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

We create a dictionary to keep track of our models

[ ]:
```
model_dict = {
    'CNN': CNN,
    'GCNN': GroupEquivariantCNN
}

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

[ ]:
```
def train_model(model_name, max_epochs=10, 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
                         gpus=1 if str(device)=="cuda:0" else 0,                        ␣
↪                 # We run on a single GPU (if possible)
                         max_epochs=max_epochs,                                          ␣
↪                 # How many epochs to train for if no patience is set
```

(continues on next page)

(continued from previous page)

```
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",␣
↪monitor="val_acc"),  # Save the best checkpoint based on the maximum val_acc recorded.␣
↪Saves only weights and not optimizer
                                    LearningRateMonitor("epoch")])
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't␣
↪need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, save_name + ".ckpt")

    if os.path.isfile(pretrained_filename):
        print(f"Found pretrained model at {pretrained_filename}, loading...")
        model = DataModule.load_from_checkpoint(pretrained_filename) # Automatically␣
↪loads the model with the saved hyperparameters
    else:
        pl.seed_everything(12) # To be reproducable
        model = DataModule(model_name=model_name, **kwargs)
        trainer.fit(model, train_loader, test_loader)
        model = DataModule.load_from_checkpoint(trainer.checkpoint_callback.best_model_
↪path) # Load best checkpoint after training

    # Test best model on test set
    val_result = trainer.test(model.to(device), test_loader, verbose=False)
    result = {"val": val_result[0]["test_acc"]}

    return model, result
```

Let's train the conventional CNN first! For this experiment, 10 epochs is sufficient for seeing the overall trend.

```
[ ]: cnn_model, cnn_results = train_model(model_name="CNN",
                                      model_hparams={"in_channels": 1,
                                                     "out_channels": 10,
                                                     "kernel_size": 5,
                                                     "num_hidden":4,
                                                     "hidden_channels":32},
                                      optimizer_name="Adam",
                                      optimizer_hparams={"lr": 1e-2,
                                                         "weight_decay": 1e-4},
                                      save_name='cnn-pretrained',
                                      max_epochs=10)
```

Next, we train the GCNN. Note that we reduce the number of channels by half to account for the increased kernel dimensionality. We do this to keep the number of trainable parameters more or less equal.

```
[ ]: gcnn_model, gcnn_results = train_model(model_name="GCNN",
                                        model_hparams={"in_channels": 1,
                                                       "out_channels": 10,
                                                       "kernel_size": 5,
                                                       "num_hidden": 4,
                                                       "hidden_channels":16, # to account␣
↪for the increase in trainable parameters due to the extra dimension in our feature␣
↪maps, remove some hidden channels.
```

(continues on next page)

(continued from previous page)

```
                                                       "group":CyclicGroup(order=4).
↪to(device)},
                                          optimizer_name="Adam",
                                          optimizer_hparams={"lr": 1e-2,
                                                             "weight_decay": 1e-4},
                                          save_name='gcnn-pretrained',
                                          max_epochs=10)
```

[ ]:
```
%%html
<!-- Some HTML code to increase font size in the following table -->
<style>
th {font-size: 120%;}
td {font-size: 120%;}
</style>
```

Let us inspect the final results from both models. As we can see the GCNN performs considerably better than the CNN. This because the GCNN implementation is invariant to rotations of 90 deg, which makes it able to recognize the handwritten digits in the test set under such rotations. Of course, since the test images are rotated by continuous rotations between 0 and 360 deg, the GCNN model still doesn't obtain perfect accuracy. How could we further improve the GCNNs generalization?

[ ]:
```
import tabulate
from IPython.display import display, HTML
all_models = [
    ("CNN", cnn_results, cnn_model),
    ("GCNN", gcnn_results, gcnn_model),
]
table = [[model_name,
          f"{100.0*model_results['val']:4.2f}%",
          "{:,}".format(sum([np.prod(p.shape) for p in model.parameters()]))]
         for model_name, model_results, model in all_models]
display(HTML(tabulate.tabulate(table, tablefmt='html', headers=["Model", "Val Accuracy",
↪"Num Parameters"])))
```

Include aboves tables in your report.

3.2 Inspecting the created feature maps

Note: This section contains couple of questions regarding the visualizations. Include an answer to each of the question in your report.

To better understand what happens inside our CNN and GCNN as we rotate an input image, let's visualize a single channel of a feature map in the second layer of the network for different rotations of an input image.

[ ]:
```
train_ds = torchvision.datasets.MNIST(root=DATASET_PATH, train=False, transform=None)

# Get an image from the test dataset.
digit, label = train_ds[123]

# Turn it into a tensor.
digit = transforms.ToTensor()(digit)
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(6, 6))
plt.imshow(digit.squeeze())
plt.title(f'Label: {label}')
plt.show()
```

[ ]:
```
plt.rcParams['figure.figsize'] = [10, 3]

# Get a set of angles by which to rotate this image.
rots = torch.linspace(0, 360 - 360/8, 8)

# Rotate the input image and push it through the normalization transform.
rot_digit = torch.stack(tuple(torchvision.transforms.functional.rotate(digit, a.item(),␣
↪torchvision.transforms.functional.InterpolationMode.BILINEAR) for a in rots))
rot_digit = torchvision.transforms.Normalize((0.1307,), (0.3081,))(rot_digit)

# Create a subfigure for every rotated input.
fig, ax = plt.subplots(1, rots.numel())

for idx, rotation in enumerate(rots):
    ax[idx].imshow(
        rot_digit[idx, :, :].squeeze()
    )
    ax[idx].set_title(f"{int(rotation)} deg")

fig.text(0.5, 0.04, 'Rotations of input image', ha='center')

plt.show()
```

[ ]:
```
# Forward it through the first few layers of the CNN.
cnn_out = cnn_model.model.first_conv(rot_digit)
cnn_out = torch.nn.functional.relu(torch.nn.functional.layer_norm(cnn_out, cnn_out.
↪shape[-3:]))
for i in range(2):
    cnn_out = cnn_model.model.convs[i](cnn_out)
    cnn_out = torch.nn.functional.relu(torch.nn.functional.layer_norm(cnn_out, cnn_out.
↪shape[-3:]))

# Let's also see what happens after we apply projection over remaining spatial dimensions.
projected_cnn_out = torch.nn.functional.adaptive_avg_pool2d(cnn_out, 1).squeeze()

# Forward it through the first few layers of the GCNN.
gcnn_out = gcnn_model.model.lifting_conv(rot_digit)
gcnn_out = torch.nn.functional.relu(torch.nn.functional.layer_norm(gcnn_out, gcnn_out.
↪shape[-4:]))
for i in range(2):
    gcnn_out = gcnn_model.model.gconvs[i](gcnn_out)
    gcnn_out = torch.nn.functional.relu(torch.nn.functional.layer_norm(gcnn_out, gcnn_
↪out.shape[-4:]))

# And let's see what happens if we apply the projection on this equivariant␣
↪representation.
projected_gcnn_out = torch.mean(gcnn_out, dim=(-3, -2, -1))
```

Let's first visualize the activations after the third convolution for a single channel of the regular CNN.

```
[ ]: plt.rcParams['figure.figsize'] = [10, 3]

     # Pick a channel to visualize
     channel_idx = 2

     # Create a subfigure for every rotated input
     fig, ax = plt.subplots(1, rots.numel())

     for idx, rotation in enumerate(rots):
         ax[idx].imshow(
             cnn_out[idx, out_channel_idx, :, :].detach().numpy()
         )
         ax[idx].set_title(f"{int(rotation)} deg")

     fig.text(0.5, 0.04, 'Rotations of input image', ha='center')

     plt.show()
```

Question: Explain what you see in the above images; how should we interpret what is happening to the feature maps as the input is rotated?

Next, we'll visualize activations after the third convolution operation for the G-CNN.

```
[ ]: plt.rcParams['figure.figsize'] = [10, 9]

     # Pick a channel to visualize
     channel_idx = 2

     # Create a subfigure for every rotated input and every group element
     fig, ax = plt.subplots(gcnn_out.shape[2], rots.numel())

     for idx, rotation in enumerate(rots):
         for group_element_idx in range(gcnn_out.shape[2]):
             ax[group_element_idx ,idx].imshow(
                 gcnn_out[idx, out_channel_idx, group_element_idx, :, :].detach().numpy()
             )
         ax[0, idx].set_title(f"{int(rotation)} deg")


     fig.text(0.5, 0.04, 'Rotations of input image', ha='center')
     fig.text(0.04, 0.5, '$H$ dimension in feature map', va='center', rotation='vertical')

     plt.show()
```

Question: Explain what you see in the above images; how should we interpret what is happening to the feature maps as the input is rotated? What happens after a 45 degree rotation? And a 90 degree rotation?

Lastly, we inspect the representations we obtain after the projection step for the CNN and GCNN. We visualize all channels. These are the representations on which the final linear layer(s) perform classification.

```
[ ]: plt.rcParams['figure.figsize'] = [8, 8]

     # Create a subfigure for every rotated input
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(rots.numel(), 1)

for idx, rotation in enumerate(rots):
    ax[idx].imshow(
        projected_cnn_out[idx, None, :].detach().numpy()
    )
    ax[idx].set_title(f"{int(rotation)} deg")
    ax[idx].set_yticks([])

fig.text(0.5, 0.04, 'Channels', ha='center')
fig.text(0.04, 0.5, 'Rotations of input image', va='center', rotation='vertical')

plt.show()
```

Question: What happens to the learned representations as the input is rotated? What consequences do you think this has on the performance of the model?

```
[ ]: plt.rcParams['figure.figsize'] = [8, 8]

# Create a subfigure for every rotated input
fig, ax = plt.subplots(rots.numel(), 1)

for idx, rotation in enumerate(rots):
    ax[idx].imshow(
        projected_gcnn_out[idx, None, :].detach().numpy()
    )
    ax[idx].set_title(f"{int(rotation)} deg")
    ax[idx].set_yticks([])

fig.text(0.5, 0.04, 'Channels', ha='center')
fig.text(0.04, 0.5, 'Rotations of input image', va='center', rotation='vertical')

plt.show()
```

Question: Explain what you see above. What does this mean for the final of input samples that are rotated?

Question: Do you notice anything about the learned representations of 45 degree vs 90 degree rotated input samples? To what can we attribute this phenomenon do you think?

3.5.5 Concluding remarks

We've seen how to implement a basic regular group convolutional network equivariant to rotations of 90 degrees. A couple suggestions of things to ponder on. Include answers to these questions in your report:

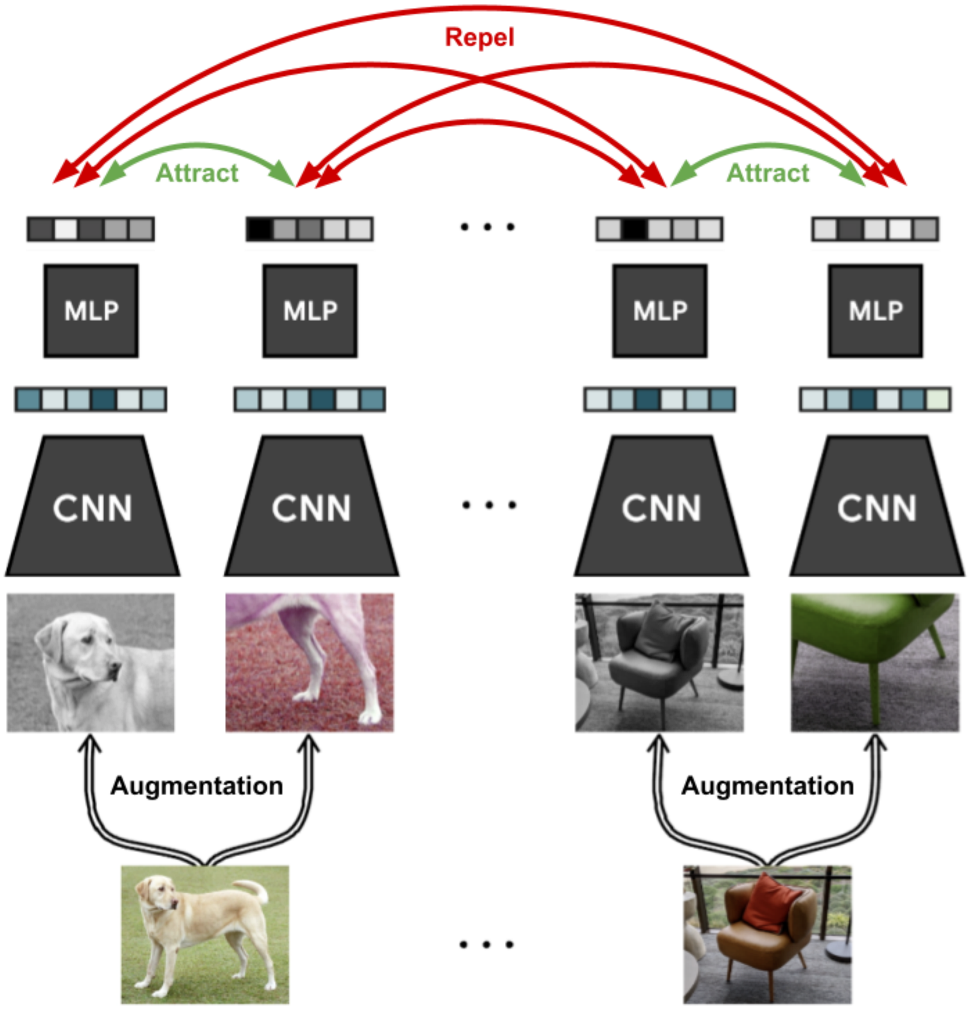
- What other groups could be interesting in computer vision problems? (these often come with their own interesting challenges).
- We used interpolation to obtain our values for transformed kernels. As mentioned, this makes our networks prone to interpolation artefacts. Would our interpolation implementation work well for other groups? Could you maybe think of other ways of defining the kernel over a continuous domain?
- Here we worked with a one-dimensional group. What would it take to implement group convolutions for multi-dimensional groups H?

3.6 Practical 5: Self-Supervised Contrastive Learning with SimCLR

Open notebook:
Authors: Phillip Lippe

In this practical, we will take a closer look at self-supervised contrastive learning. Self-supervised learning, or also sometimes called unsupervised learning, describes the scenario where we have given input data, but no accompanying labels to train in a classical supervised way. However, this data still contains a lot of information from which we can learn: how are the images different from each other? What patterns are descriptive for certain images? Can we cluster the images? And so on. Methods for self-supervised learning try to learn as much as possible from the data alone, so it can quickly be finetuned for a specific classification task. The benefit of self-supervised learning is that a large dataset can often easily be obtained. For instance, if we want to train a vision model on semantic segmentation for autonomous driving, we can collect large amounts of data by simply installing a camera in a car, and driving through a city for an hour. In contrast, if we would want to do supervised learning, we would have to manually label all those images before training a model. This is extremely expensive, and would likely take a couple of months to manually label the same amount of data. Further, self-supervised learning can provide an alternative to transfer learning from models pretrained on ImageNet since we could pretrain a model on a specific dataset/situation, e.g. traffic scenarios for autonomous driving.

Within the last two years, a lot of new approaches have been proposed for self-supervised learning, in particular for images, that have resulted in great improvements over supervised models when few labels are available. The subfield that we will focus on in this tutorial is contrastive learning. Contrastive learning is motivated by the question mentioned above: how are images different from each other? Specifically, contrastive learning methods train a model to cluster an image and its slightly augmented version in latent space, while the distance to other images should be maximized. A very recent and simple method for this is SimCLR, which is visualized below (figure credit - Ting Chen et al.).



The general setup is that we are given a dataset of images without any labels, and want to train a model on this data such that it can quickly adapt to any image recognition task afterward. During each training iteration, we sample a batch of images as usual. For each image, we create two versions by applying data augmentation techniques like cropping, Gaussian noise, blurring, etc. An example of such is shown on the left with the image of the dog. We will go into the details and effects of the chosen augmentation techniques later. On those images, we apply a CNN like ResNet and obtain as output a 1D feature vector on which we apply a small MLP. The output features of the two augmented images are then trained to be close to each other, while all other images in that batch should be as different as possible. This way, the model has to learn to recognize the content of the image that remains unchanged under the data augmentations, such as objects which we usually care about in supervised tasks.

We will now implement this framework ourselves and discuss further details along the way. Let's first start with importing our standard libraries below:

[1]:
```
## Standard libraries
import os
import numpy as np
import random
import math
import json
from functools import partial
from PIL import Image
```

(continues on next page)

(continued from previous page)

```
## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set()

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

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

## Torchvision
import torchvision
from torchvision.datasets import CIFAR10, STL10
from torchvision import transforms

# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by␣
↪default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.6
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Import tensorboard
%load_ext tensorboard

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/practical5"

# Setting the seed
pl.seed_everything(42)
# In this notebook, we use data loaders with heavier computational processing. It is␣
↪recommended to use as many
# workers as possible in a data loader, which corresponds to the number of CPU cores
NUM_WORKERS = os.cpu_count()

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

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

(continues on next page)

(continued from previous page)

```
print("Number of workers:", NUM_WORKERS)
```

```
Global seed set to 42
```

```
Device: cuda:0
Number of workers: 16
```

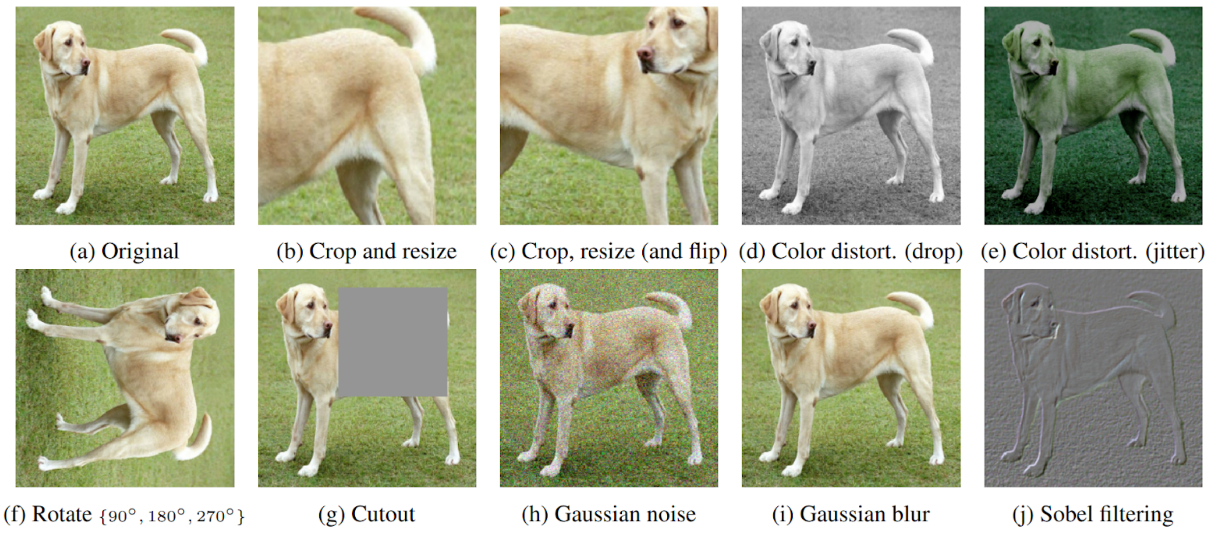
3.6.1 Data Augmentation for Contrastive Learning

We will start our exploration of contrastive learning by discussing the effect of different data augmentation techniques, and how we can implement an efficient data loader for such. To allow efficient training, we need to prepare the data loading such that we sample two different, random augmentations for each image in the batch. The easiest way to do this is by creating a transformation that, when being called, applies a set of data augmentations to an image twice. This is implemented in the class `ContrastiveTransformations` below:

[2]:
```
class ContrastiveTransformations(object):

    def __init__(self, base_transforms, n_views=2):
        self.base_transforms = base_transforms
        self.n_views = n_views

    def __call__(self, x):
        return [self.base_transforms(x) for i in range(self.n_views)]
```

The contrastive learning framework can easily be extended to have more *positive* examples by sampling more than two augmentations of the same image. However, the most efficient training is usually obtained by using only two.

Next, we can look at the specific augmentations we want to apply. The choice of the data augmentation to use is the most crucial hyperparameter in SimCLR since it directly affects how the latent space is structured, and what patterns might be learned from the data. Let's first take a look at some of the most popular data augmentations (figure credit - Ting Chen and Geoffrey Hinton):


(a) Original
(b) Crop and resize
(c) Crop, resize (and flip)
(d) Color distort. (drop)
(e) Color distort. (jitter)
(f) Rotate {90°, 180°, 270°}
(g) Cutout
(h) Gaussian noise
(i) Gaussian blur
(j) Sobel filtering

All of them can be used, but it turns out that two augmentations stand out in their importance: crop-and-resize, and color distortion. Interestingly, however, they only lead to strong performance if they have been used together as discussed by Ting Chen et al. in their SimCLR paper. When performing randomly cropping and resizing, we can distinguish between two situations: (a) cropped image A provides a local view of cropped image B, or (b) cropped images C and D show neighboring views of the same image (figure credit - Ting Chen and Geoffrey Hinton).

While situation (a) requires the model to learn some sort of scale invariance to make crops A and B similar in latent space, situation (b) is more challenging since the model needs to recognize an object beyond its limited view. However, without color distortion, there is a loophole that the model can exploit, namely that different crops of the same image usually look very similar in color space. Consider the picture of the dog above. Simply from the color of the fur and the green color tone of the background, you can reason that two patches belong to the same image without actually recognizing the dog in the picture. In this case, the model might end up focusing only on the color histograms of the images, and ignore other more generalizable features. If, however, we distort the colors in the two patches randomly and independently of each other, the model cannot rely on this simple feature anymore. Hence, by combining random cropping and color distortions, the model can only match two patches by learning generalizable representations.

Overall, for our experiments, we apply a set of 5 transformations following the original SimCLR setup: random horizontal flip, crop-and-resize, color distortion, random grayscale, and gaussian blur. In comparison to the original implementation, we reduce the effect of the color jitter slightly (0.5 instead of 0.8 for brightness, contrast, and saturation, and 0.1 instead of 0.2 for hue). In our experiments, this setting obtained better performance and was faster and more stable to train. If, for instance, the brightness scale highly varies in a dataset, the original settings can be more beneficial since the model can't rely on this information anymore to distinguish between images.

[3]:
```
contrast_transforms = transforms.Compose([transforms.RandomHorizontalFlip(),
                                          transforms.RandomResizedCrop(size=32, scale=(0.
↪25, 1.0)),
                                          transforms.RandomApply([
                                              transforms.ColorJitter(brightness=0.5,
                                                                     contrast=0.5,
                                                                     saturation=0.5,
                                                                     hue=0.1)
                                          ], p=0.8),
                                          transforms.RandomGrayscale(p=0.2),
                                          transforms.GaussianBlur(kernel_size=3),
                                          transforms.ToTensor(),
                                          transforms.Normalize((0.5,), (0.5,))
                                         ])
```

After discussing the data augmentation techniques, we can now focus on the dataset. In this tutorial, we will use the STL10 dataset, which, similarly to CIFAR10, contains images of 10 classes: airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck. However, the images have a higher resolution, namely 96×96 pixels, and we are only provided with 500 labeled images per class. Additionally, we have a much larger set of $100,000$ unlabeled images which are similar to the training images but are sampled from a wider range of animals and vehicles. This makes the dataset ideal to showcase the benefits that self-supervised learning offers. For this practical, however, to reduce the computationally complexity, we will downscale the images back to 32×32 pixels.

Luckily, the STL10 dataset is provided through torchvision. Keep in mind, however, that since this dataset is relatively large and has a considerably higher resolution than CIFAR10, it requires more disk space (~3GB) and takes a bit of time to download. For our initial discussion of self-supervised learning and SimCLR, we will create two data loaders with our contrastive transformations above: the `unlabeled_data` will be used to train our model via contrastive learning, and `train_data_contrast` will be used as a validation set in contrastive learning.

[4]:
```
unlabeled_data = STL10(root=DATASET_PATH, split='unlabeled', download=True,
                       transform=ContrastiveTransformations(contrast_transforms, n_
↪views=2))
train_data_contrast = STL10(root=DATASET_PATH, split='train', download=True,
                            transform=ContrastiveTransformations(contrast_transforms, n_
↪views=2))
```
(continues on next page)

(continued from previous page)

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

[5]:
```
# Downscale images to 32x32 directly in arrays to save RAM and data loading computation
def downscale_dataset(dataset):
    data = dataset.data
    num_imgs = data.shape[0]
    new_data = np.zeros((num_imgs, data.shape[1], 32, 32), dtype=data.dtype)
    for i in tqdm(range(0, num_imgs, 100)):
        new_data[i:i+100] = transforms.functional.resize(torch.from_numpy(data[i:i+100]).
↪float(), size=[32, 32]).to(torch.uint8).numpy()
    dataset.data = new_data

downscale_dataset(unlabeled_data)
downscale_dataset(train_data_contrast)
```

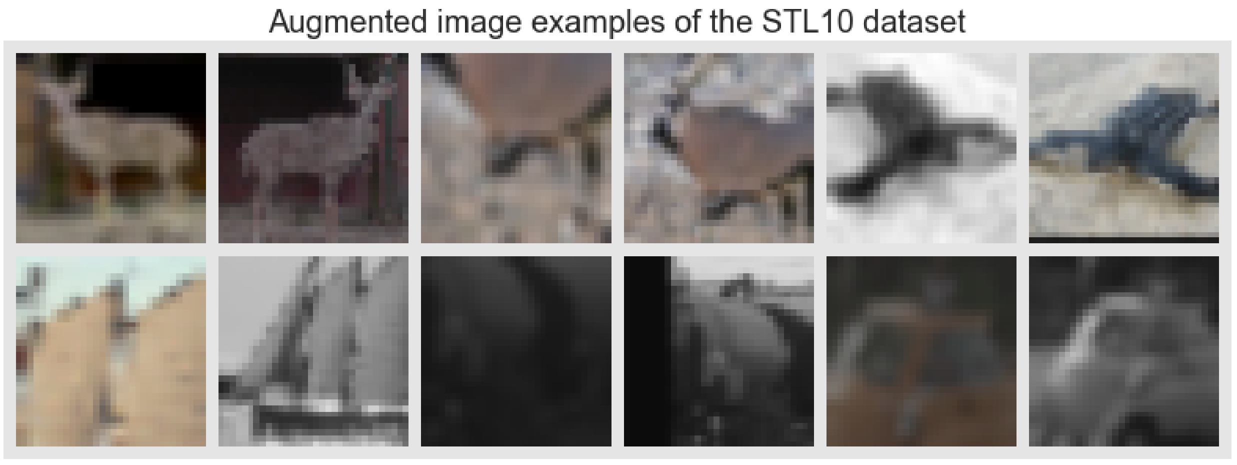
```
  0%|          | 0/1000 [00:00<?, ?it/s]
```

```
  0%|          | 0/50 [00:00<?, ?it/s]
```

Finally, before starting with our implementation of SimCLR, let’s look at some example image pairs sampled with our augmentations:

[6]:
```
# Visualize some examples
pl.seed_everything(42)
NUM_IMAGES = 6
imgs = torch.stack([img for idx in range(NUM_IMAGES) for img in unlabeled_data[idx][0]],␣
↪dim=0)
img_grid = torchvision.utils.make_grid(imgs, nrow=6, normalize=True, pad_value=0.9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(14,7))
plt.title('Augmented image examples of the STL10 dataset', fontsize=20)
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

```
Global seed set to 42
```



We see the wide variety of our data augmentation, including randomly cropping, grayscaling, gaussian blur, and color distortion. Thus, it remains a challenging task for the model to match two, independently augmented patches of the same image.

3.6.2 Part 1: SimCLR implementation

Using the data loader pipeline above, we can now implement SimCLR. At each iteration, we get for every image x two differently augmented versions, which we refer to as $\tilde{x}_i$ and $\tilde{x}_j$. Both of these images are encoded into a one-dimensional feature vector, between which we want to maximize similarity which minimizes it to all other images in the batch. The encoder network is split into two parts: a base encoder network $f(\cdot)$, and a projection head $g(\cdot)$. The base network is usually a deep CNN as we have seen in e.g. Tutorial 5 before, and is responsible for extracting a representation vector from the augmented data examples. In our experiments, we will use the common ResNet-18 architecture as $f(\cdot)$, and refer to the output as $f(\tilde{x}_i) = h_i$. The projection head $g(\cdot)$ maps the representation h into a space where we apply the contrastive loss, i.e., compare similarities between vectors. It is often chosen to be a small MLP with non-linearities, and for simplicity, we follow the original SimCLR paper setup by defining it as a two-layer MLP with ReLU activation in the hidden layer. Note that in the follow-up paper, SimCLRv2, the authors mention that larger/wider MLPs can boost the performance considerably. This is why we apply an MLP with four times larger hidden dimensions, but deeper MLPs showed to overfit on the given dataset. The general setup is visualized below (figure credit - Ting Chen et al.):

After finishing the training with contrastive learning, we will remove the projection head $g(\cdot)$, and use $f(\cdot)$ as a pre-trained feature extractor. The representations z that come out of the projection head $g(\cdot)$ have been shown to perform worse than those of the base network $f(\cdot)$ when finetuning the network for a new task. This is likely because the representations z are trained to become invariant to many features like the color that can be important for downstream tasks. Thus, $g(\cdot)$ is only needed for the contrastive learning stage.

Let's first start by implementing a Base Network which will represent function $f(\cdot)$. Usually, you would use very large, powerful networks like a deep ResNet, but these are very expensive to train. To reduce the computational cost and make it possible to reasonable train the models on Google Colab, we provide a very simple CNN here:

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

         def __init__(self, num_input_channels, c_hid, output_dim):
             """
             Inputs:
                 - num_input_channels : Number of input channels of the image. For CIFAR,␣
     ↪this parameter is 3
```

(continues on next page)

(continued from previous page)

```
            - c_hid : Number of channels we use in the first convolutional layers.␣
↪Deeper layers might use a duplicate of it.
            - output_dim : Dimensionality of the final latent representation
        """
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(num_input_channels, c_hid, kernel_size=3, padding=1, stride=2), #␣
↪32x32 => 16x16
            nn.BatchNorm2d(c_hid),
            nn.SiLU(),
            nn.Conv2d(c_hid, c_hid, kernel_size=3, padding=1),
            nn.BatchNorm2d(c_hid),
            nn.SiLU(),
            nn.Conv2d(c_hid, 2*c_hid, kernel_size=3, padding=1, stride=2), # 16x16 => 8x8
            nn.BatchNorm2d(2*c_hid),
            nn.SiLU(),
            nn.Conv2d(2*c_hid, 2*c_hid, kernel_size=3, padding=1),
            nn.BatchNorm2d(2*c_hid),
            nn.SiLU(),
            nn.Conv2d(2*c_hid, 2*c_hid, kernel_size=3, padding=1, stride=2), # 8x8 => 4x4
            nn.BatchNorm2d(2*c_hid),
            nn.SiLU(),
            nn.Flatten(), # Image grid to single feature vector
            nn.Linear(2*16*c_hid, output_dim)
        )

    def forward(self, x):
        return self.net(x)
```

Now that the architecture is described, let's take a closer look at how we train the model. As mentioned before, we want to maximize the similarity between the representations of the two augmented versions of the same image, i.e., z_i and z_j in the figure above, while minimizing it to all other examples in the batch. SimCLR thereby applies the InfoNCE loss, originally proposed by Aaron van den Oord et al. for contrastive learning. In short, the InfoNCE loss compares the similarity of z_i and z_j to the similarity of z_i to any other representation in the batch by performing a softmax over the similarity values. The loss can be formally written as:

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k\neq i]} \exp(\text{sim}(z_i, z_k)/\tau)} = -\text{sim}(z_i, z_j)/\tau + \log\left[\sum_{k=1}^{2N} \mathbb{1}_{[k\neq i]} \exp(\text{sim}(z_i, z_k)/\tau)\right]$$

The function sim is a similarity metric, and the hyperparameter τ is called temperature determining how peaked the distribution is. Since many similarity metrics are bounded, the temperature parameter allows us to balance the influence of many dissimilar image patches versus one similar patch. The similarity metric that is used in SimCLR is cosine similarity, as defined below:

$$\text{sim}(z_i, z_j) = \frac{z_i^\top \cdot z_j}{||z_i|| \cdot ||z_j||}$$

The maximum cosine similarity possible is 1, while the minimum is -1. In general, we will see that the features of two different images will converge to a cosine similarity around zero since the minimum, -1, would require z_i and z_j to be in the exact opposite direction in all feature dimensions, which does not allow for great flexibility.

Finally, now that we have discussed all details, let's implement SimCLR below as a PyTorch Lightning module:

```
[ ]: class SimCLR(pl.LightningModule):

         def __init__(self, hidden_dim, lr, temperature, weight_decay, max_epochs=500):
             super().__init__()
             self.save_hyperparameters()
             assert self.hparams.temperature > 0.0, 'The temperature must be a positive float!
     ↪'
             # TODO: Setup the Base Network
             raise NotImplementedError

         def configure_optimizers(self):
             optimizer = optim.AdamW(self.parameters(),
                                     lr=self.hparams.lr,
                                     weight_decay=self.hparams.weight_decay)
             lr_scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer,
                                                                 T_max=self.hparams.max_
     ↪epochs,
                                                                 eta_min=self.hparams.lr/50)
             return [optimizer], [lr_scheduler]

         def info_nce_loss(self, batch, mode='train'):
             imgs, _ = batch  # we do not need the labels here
             # imgs is a list of length 2, where imgs[0][i] and imgs[1][i] are the positive␣
     ↪pairs

             # TODO: Calculate the contrastive loss of SimCLR. Try to be as efficient as␣
     ↪possible
             # Hint: if you add imgs into a batch where over dimension 0, you have [imgs[0],
     ↪imgs[1]],
             # the positive pair for an image at position i is always at (i + batch_size) %␣
     ↪(2 * batch_size)
             # Can you create a mask to find the positive element for each batch element?
             raise NotImplementedError

             # TODO: Log the loss and the top-1 and top-5 accuracy as how often the most␣
     ↪similar image was the positive
             # You can also split this part into another function if you prefer
             raise NotImplementedError
             self.log(mode+'_acc_top1', ...)
             self.log(mode+'_acc_top5', ...)

             return loss

         def training_step(self, batch, batch_idx):
             return self.info_nce_loss(batch, mode='train')

         def validation_step(self, batch, batch_idx):
             self.info_nce_loss(batch, mode='val')
```

```
[ ]: # TODO: Create some test cases yourself to check the INFO-NCE loss and the logging!
     raise NotImplementedError
```

Alternatively to performing the validation on the contrastive learning loss as well, we could also take a simple, small downstream task, and track the performance of the base network $f(\cdot)$ on that. However, in this tutorial, we will restrict ourselves to the STL10 dataset where we use the task of image classification on STL10 as our test task.

Training

Now that we have implemented SimCLR and the data loading pipeline, we are ready to train the model. We will use the same training function setup as usual. For saving the best model checkpoint, we track the metric `val_acc_top5`, which describes how often the correct image patch is within the top-5 most similar examples in the batch. This is usually less noisy than the top-1 metric, making it a better metric to choose the best model from.

```
[ ]: def train_simclr(batch_size, max_epochs=500, **kwargs):
         trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, 'SimCLR'),
                              gpus=1 if str(device)=='cuda:0' else 0,
                              max_epochs=max_epochs,
                              callbacks=[ModelCheckpoint(save_weights_only=True, mode='max',␣
     ↪monitor='val_acc_top5'),
                                         LearningRateMonitor('epoch')],
                              check_val_every_n_epoch=5)
         trainer.logger._default_hp_metric = None # Optional logging argument that we don't␣
     ↪need

         # Check whether pretrained model exists. If yes, load it and skip training
         pretrained_filename = os.path.join(CHECKPOINT_PATH, 'SimCLR.ckpt')
         if os.path.isfile(pretrained_filename):
             print(f'Found pretrained model at {pretrained_filename}, loading...')
             model = SimCLR.load_from_checkpoint(pretrained_filename) # Automatically loads␣
     ↪the model with the saved hyperparameters
         else:
             train_loader = data.DataLoader(unlabeled_data, batch_size=batch_size,␣
     ↪shuffle=True,
                                            drop_last=True, pin_memory=True, num_workers=NUM_
     ↪WORKERS)
             val_loader = data.DataLoader(train_data_contrast, batch_size=batch_size,␣
     ↪shuffle=False,
                                           drop_last=False, pin_memory=True, num_workers=NUM_
     ↪WORKERS)
             pl.seed_everything(42) # To be reproducable
             model = SimCLR(max_epochs=max_epochs, **kwargs)
             trainer.fit(model, train_loader, val_loader)
             model = SimCLR.load_from_checkpoint(trainer.checkpoint_callback.best_model_path)
     ↪# Load best checkpoint after training

         return model
```

A common observation in contrastive learning is that the larger the batch size, the better the models perform. A larger batch size allows us to compare each image to more negative examples, leading to overall smoother loss gradients, but a batch size of 256 is sufficient here. Again, for a first run, you can use 10 epochs, but try to increase the number of epochs for a final run.

```
[ ]: simclr_model = train_simclr(batch_size=256,
                                 hidden_dim=128,
                                 lr=5e-4,
                                 temperature=0.07,
```

(continues on next page)

(continued from previous page)

```
                                  weight_decay=1e-4,
                                  max_epochs=10)
```

To get an intuition of how training with contrastive learning behaves, we can take a look at the TensorBoard below:

```
[ ]: %tensorboard --logdir ../saved_models/practical5/SimCLR/
```

In your report, show the top-1 and top-5 accuracy validation curves. Discuss the overall performance and the training speed. Is the model already converged? What does the final performance of the model imply about the learned feature space?

3.6.3 Part 2: Logistic Regression

After we have trained our model via contrastive learning, we can deploy it on downstream tasks and see how well it performs with little data. A common setup, which also verifies whether the model has learned generalized representations, is to perform Logistic Regression on the features. In other words, we learn a single, linear layer that maps the representations to a class prediction. Since the base network $f(\cdot)$ is not changed during the training process, the model can only perform well if the representations of h describe all features that might be necessary for the task. Further, we do not have to worry too much about overfitting since we have very few parameters that are trained. Hence, we might expect that the model can perform well even with very little data.

First, let's implement a simple Logistic Regression setup for which we assume that the images already have been encoded in their feature vectors. If very little data is available, it might be beneficial to dynamically encode the images during training so that we can also apply data augmentations. However, the way we implement it here is much more efficient and can be trained within a few seconds. Further, using data augmentations did not show any significant gain in this simple setup.

```
[ ]: class LogisticRegression(pl.LightningModule):

         def __init__(self, feature_dim, num_classes, lr, weight_decay, max_epochs=100):
             super().__init__()
             self.save_hyperparameters()
             # Mapping from representation h to classes
             # TODO: Initialize logistic regression model
             raise NotImplementedError

         def configure_optimizers(self):
             optimizer = optim.AdamW(self.parameters(),
                                     lr=self.hparams.lr,
                                     weight_decay=self.hparams.weight_decay)
             lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer,
                                                           milestones=[int(self.hparams.max_
↪epochs*0.6),
                                                                       int(self.hparams.max_
↪epochs*0.8)],
                                                           gamma=0.1)
             return [optimizer], [lr_scheduler]

         def _calculate_loss(self, batch, mode='train'):
             # TODO: Calculate classification loss for logistic regression model
             raise NotImplementedError
```

(continues on next page)

(continued from previous page)

```
    def training_step(self, batch, batch_idx):
        return self._calculate_loss(batch, mode='train')

    def validation_step(self, batch, batch_idx):
        self._calculate_loss(batch, mode='val')

    def test_step(self, batch, batch_idx):
        self._calculate_loss(batch, mode='test')
```

The data we use is the training and test set of STL10. The training contains 500 images per class, while the test set has 800 images per class.

```
[ ]: img_transforms = transforms.Compose([transforms.ToTensor(),
                                         transforms.Normalize((0.5,), (0.5,))])

     train_img_data = STL10(root=DATASET_PATH, split='train', download=True,
                            transform=img_transforms)
     test_img_data = STL10(root=DATASET_PATH, split='test', download=True,
                           transform=img_transforms)
     downscale_dataset(train_img_data)
     downscale_dataset(test_img_data)

     print("Number of training examples:", len(train_img_data))
     print("Number of test examples:", len(test_img_data))
```

Next, we implement a small function to encode all images in our datasets. The output representations are then used as inputs to the Logistic Regression model.

```
[ ]: @torch.no_grad()
     def prepare_data_features(model, dataset):
         # TODO: Obtain the feature representation for all images in the dataset
         raise NotImplementedError

         # Return a new dataset with the image features and labels
         return data.TensorDataset(feats, labels)
```

Let’s apply the function to both training and test set below.

```
[ ]: train_feats_simclr = prepare_data_features(simclr_model, train_img_data)
     test_feats_simclr = prepare_data_features(simclr_model, test_img_data)
```

Finally, we can write a training function as usual. We evaluate the model on the test set every 10 epochs to allow early stopping, but the low frequency of the validation ensures that we do not overfit too much on the test set.

```
[ ]: def train_logreg(batch_size, train_feats_data, test_feats_data, model_suffix, max_
     ↪epochs=100, **kwargs):
         trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH,
     ↪"LogisticRegression"),
                              gpus=1 if str(device)=="cuda:0" else 0,
                              max_epochs=max_epochs,
                              callbacks=[ModelCheckpoint(save_weights_only=True, mode='max',␣
     ↪monitor='val_acc'),
                                         LearningRateMonitor("epoch")],
```

(continues on next page)

(continued from previous page)

```
                         progress_bar_refresh_rate=0,
                         check_val_every_n_epoch=10)
    trainer.logger._default_hp_metric = None

    # Data loaders
    train_loader = data.DataLoader(train_feats_data, batch_size=batch_size, shuffle=True,
                                   drop_last=False, pin_memory=True, num_workers=0)
    test_loader = data.DataLoader(test_feats_data, batch_size=batch_size, shuffle=False,
                                  drop_last=False, pin_memory=True, num_workers=0)

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, f"LogisticRegression_{model_
↪suffix}.ckpt")
    if os.path.isfile(pretrained_filename):
        print(f"Found pretrained model at {pretrained_filename}, loading...")
        model = LogisticRegression.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything(42)  # To be reproducable
        model = LogisticRegression(**kwargs)
        trainer.fit(model, train_loader, test_loader)
        model = LogisticRegression.load_from_checkpoint(trainer.checkpoint_callback.best_
↪model_path)

    # Test best model on train and validation set
    train_result = trainer.test(model, train_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"train": train_result[0]["test_acc"], "test": test_result[0]["test_acc"]}

    return model, result
```

Despite the training dataset of STL10 already only having 500 labeled images per class, we will perform experiments with even smaller datasets. Specifically, we train a Logistic Regression model for datasets with only 10, 20, 50, 100, 200, and all 500 examples per class. This gives us an intuition on how well the representations learned by contrastive learning can be transfered to a image recognition task like this classification. First, let's define a function to create the intended sub-datasets from the full training set:

```
[ ]: def get_smaller_dataset(original_dataset, num_imgs_per_label):
         # TODO: Return dataset with the first N images per label
         raise NotImplementedError
```

Next, let's run all models. Despite us training 6 models, this cell could be run within a minute or two without the pretrained models.

```
[ ]: # TODO: Run the logistic regression on datasets of 10, 20, 50, 100, 200, and 500 labeled␣
     ↪examples
     raise NotImplementedError
```

Finally, let's plot the results.

```
[ ]: # TODO: Plot the results
     raise NotImplementedError
```

Add the plot in your report and discuss the overall performance as well as the trend you see in the plot. Where do you see the biggest jump in performance? Is the performance already saturating?

3.6.4 Part 3: Baseline

As a baseline to our results above, we will train the BaseNetwork with random initialization on the labeled training set of STL10. The results will give us an indication of the advantages that contrastive learning on unlabeled data has compared to using only supervised training. First, let's implement it below.

```
[ ]: class Baseline(pl.LightningModule):

         def __init__(self, num_classes, lr, weight_decay, max_epochs=100):
             super().__init__()
             self.save_hyperparameters()
             # TODO: Initialize a base network
             raise NotImplementedError

         def configure_optimizers(self):
             optimizer = optim.AdamW(self.parameters(),
                                     lr=self.hparams.lr,
                                     weight_decay=self.hparams.weight_decay)
             lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer,
                                                           milestones=[int(self.hparams.max_
     ↪epochs*0.7),
                                                                       int(self.hparams.max_
     ↪epochs*0.9)],
                                                           gamma=0.1)
             return [optimizer], [lr_scheduler]

         def _calculate_loss(self, batch, mode='train'):
             # TODO: Calculate classification loss and accuracy
             raise NotImplementedError

         def training_step(self, batch, batch_idx):
             return self._calculate_loss(batch, mode='train')

         def validation_step(self, batch, batch_idx):
             self._calculate_loss(batch, mode='val')

         def test_step(self, batch, batch_idx):
             self._calculate_loss(batch, mode='test')
```

It is clear that the ResNet easily overfits on the training data since its parameter count is more than 1000 times larger than the dataset size. To make the comparison to the contrastive learning models fair, we apply data augmentations similar to the ones we used before: horizontal flip, crop-and-resize, grayscale, and gaussian blur. Color distortions as before are not used because the color distribution of an image showed to be an important feature for the classification. Hence, we observed no noticeable performance gains when adding color distortions to the set of augmentations. Similarly, we restrict the resizing operation before cropping to the max. 125% of its original resolution, instead of 1250% as done in SimCLR. This is because, for classification, the model needs to recognize the full object, while in contrastive learning, we only want to check whether two patches belong to the same image/object. Hence, the chosen augmentations below are overall weaker than in the contrastive learning case.

```
[ ]: train_transforms = transforms.Compose([transforms.RandomHorizontalFlip(),
                                            transforms.RandomResizedCrop(size=32, scale=(0.8,␣
     ↪1.0)),
```

(continues on next page)

(continued from previous page)

```
                                              transforms.RandomGrayscale(p=0.2),
                                              transforms.GaussianBlur(kernel_size=3, sigma=(0.1,
↪ 0.5)),
                                              transforms.ToTensor(),
                                              transforms.Normalize((0.5,), (0.5,))
                                              ])

train_img_aug_data = STL10(root=DATASET_PATH, split='train', download=True,
                           transform=train_transforms)
downscale_dataset(train_img_aug_data)
```

The training function for the ResNet is almost identical to the Logistic Regression setup. Note that we allow the ResNet to perform validation every 2 epochs to also check whether the model overfits strongly in the first iterations or not.

[ ]:
```
def train_baseline(batch_size, max_epochs=100, **kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "ResNet"),
                         gpus=1 if str(device)=="cuda:0" else 0,
                         max_epochs=max_epochs,
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",␣
↪monitor="val_acc"),
                                    LearningRateMonitor("epoch")],
                         progress_bar_refresh_rate=1,
                         check_val_every_n_epoch=10)
    trainer.logger._default_hp_metric = None

    # Data loaders
    train_loader = data.DataLoader(train_img_aug_data, batch_size=batch_size,␣
↪shuffle=True,
                                   drop_last=True, pin_memory=True, num_workers=NUM_
↪WORKERS)
    test_loader = data.DataLoader(test_img_data, batch_size=batch_size, shuffle=False,
                                  drop_last=False, pin_memory=True, num_workers=NUM_
↪WORKERS)

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "ResNet.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model at %s, loading..." % pretrained_filename)
        model = Baseline.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything(42) # To be reproducable
        model = Baseline(**kwargs)
        trainer.fit(model, train_loader, test_loader)
        model = Baseline.load_from_checkpoint(trainer.checkpoint_callback.best_model_
↪path)

    # Test best model on validation set
    train_result = trainer.test(model, train_loader, verbose=False)
    val_result = trainer.test(model, test_loader, verbose=False)
    result = {"train": train_result[0]["test_acc"], "test": val_result[0]["test_acc"]}

    return model, result
```

Finally, let's train the model and check its results. For a first run, use 10 epochs, but to obtain final results, try to train the model on more epochs.

```
[ ]: baseline_model, baseline_result = train_baseline(batch_size=64,
                                                      num_classes=10,
                                                      lr=1e-3,
                                                      weight_decay=2e-4,
                                                      max_epochs=10)
```

```
[ ]: print(f"Accuracy on training set: {baseline_result['train']:.2%}")
     print(f"Accuracy on test set: {baseline_result['test']:.2%}")
```

In your report, note the results you have obtained from this baseline and compare it to the logistic regression model in Part 2. What do you see? What do the results imply?

3.6.5 Conclusion

In this tutorial, we have discussed self-supervised contrastive learning and implemented SimCLR as an example method. We have applied it to the STL10 dataset and showed that it can learn generalizable representations that we can use to train simple classification models. Besides the discussed hyperparameters, the size of the model seems to be important in contrastive learning as well. If a lot of unlabeled data is available, larger models can achieve much stronger results and come close to their supervised baselines. Further, there are also approaches for combining contrastive and supervised learning, leading to performance gains beyond supervision (see Khosla et al.). Moreover, contrastive learning is not the only approach to self-supervised learning that has come up in the last two years and showed great results. Other methods include distillation-based methods like BYOL and redundancy reduction techniques like Barlow Twins. There is a lot more to explore in the self-supervised domain.

References

[1] Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. (2020). A simple framework for contrastive learning of visual representations. In International conference on machine learning (pp. 1597-1607). PMLR. (link)

[2] Chen, T., Kornblith, S., Swersky, K., Norouzi, M., and Hinton, G. (2020). Big self-supervised models are strong semi-supervised learners. NeurIPS 2021 (link).

[3] Oord, A. V. D., Li, Y., and Vinyals, O. (2018). Representation learning with contrastive predictive coding. arXiv preprint arXiv:1807.03748. (link)

[4] Grill, J.B., Strub, F., Altché, F., Tallec, C., Richemond, P.H., Buchatskaya, E., Doersch, C., Pires, B.A., Guo, Z.D., Azar, M.G. and Piot, B. (2020). Bootstrap your own latent: A new approach to self-supervised learning. arXiv preprint arXiv:2006.07733. (link)

[5] Khosla, P., Teterwak, P., Wang, C., Sarna, A., Tian, Y., Isola, P., Maschinot, A., Liu, C. and Krishnan, D. (2020). Supervised contrastive learning. arXiv preprint arXiv:2004.11362. (link)

[6] Zbontar, J., Jing, L., Misra, I., LeCun, Y. and Deny, S. (2021). Barlow twins: Self-supervised learning via redundancy reduction. arXiv preprint arXiv:2103.03230. (link)