HPML blogposts

Release 0.1

Bryan Cardenas Guevara

Dec 07, 2022

TUTORIALS:

CHAPTER

ONE

SURF HPML DOCUMENTATION!

The high performance machine learning group at SURF facilitates efficient deep learning usage on the Dutch national supercomputer. Here we provide the documentation for our tutorials, presentations and blogposts!

Our group has in-house expertise on several topics including computational histopathology, GPU programming, physics informed DL, multi-modal Learning and large language modelling! For more information please go to this post about ML in HPC environments!

Warning: This project is currently under verocious development.

This GitHub template includes tutorials, blogposts and slides.

SURF Website: https://www.surf.nl/
Repository: https://github.com/Cryptheon/hpml-surf
Author: Bryan Cardenas Guevara

CHAPTER

TWO

CONTENTS:

2.1 Large Language Models on Snellius

In this post we demonstrate how to run a large language model on Snellius. The main purpose of this small experiment was explorative in nature; to which extent can we perform generation or latent extraction on Snellius? How much compute is needed for a single prompt?

We will mainly discuss:

1. How we got it working on Snellius.
2. how to run it.
3. a few examples.

The main repository and the tested downloaded models can be found on Snellius under `/projects/0/hpmlprjs/GALACTICA/`. For now, we named it GALACTICA as it was solely intended for the new Meta's Galactica scientific models. Although, we could use any causal language model uploaded to the Huggingface Huggingface hub. More specifically, any model that can be loaded using `AutoTokenizer` and `AutoModelForCausalLM`. Do note

Note: Testing is still necessary as some models break under specific `PyTorch`, `transformers` or `DeepSpeed` versions.

Warning: This blog is mainly intended for the HPML members for now. A more public version is coming soon, GPUs near you.

For now we have tested four different language models:

- BLOOM, a multi-language language model (40+ languages)
- Galactica 6.7b, the galactic models are a family of LMs trained solely on scientific data
- OPT-30b, LM trained on 800GB of text data (180B tokens).
- GPT-NeoX-20b, LM trained by EleutherAI on The Pile

These models all have one clearly overlapping feature; they are decoder-transformers similar in shape to GPT-2 and GPT-3. It stands to overemphasize that each has their own qualities and desired properties and as such, it would be beneficial to keep a few of these models on Snellius as the need arises.

2.1.1 1. How we got it running on Snellius

We will see how we downloaded and loaded the model for generation.

Let's take galactica uploaded by Meta on Huggingface as an example. The sharded model can be found under `files and versions`. We first need to have git lfs installed to be able to download these files on our disk.

We can use

```
git lfs clone https://huggingface.co/facebook/galactica-6.7b/
```

or we can just use `git clone` in this case.

Note: Using git lfs for larger language models such as BLOOM-176b, we would first be downloading specific binaries that would need to be constructed afterwards by running `git lfs checkout`.

Let's look at how to load this model using Hugginface. We use `transformers==4.21` and `accelerate`, which is Hugginface's own distributed computing framework that will make our lives easier for now.

Loading the Tokenizer and Model

To avoid bloat and confusion we show the important parts only, please take a look at `./GALACTICA/lm_gen.py` for more details.

```
tokenizer = AutoTokenizer.from_pretrained(args.model_path)

kwargs = dict(device_map="auto", load_in_8bit=False)

model = AutoModelForCausalLM.from_pretrained(args.model_path, **kwargs)
```

Here we see how we prepare the tokenizer and load the model for the given model path. In this case we use `/GALACTICA/langauge_models/galactica-6.7b`, in which we can find the model weights and the tokenizer. In kwargs we can see `device_map="auto"` and `load_in_8bit=False`.

With the former we tell the accelerate framework to load the checkpoint automatically. The accelerate framework enables us to run a model in any distributed configuration, it supports sharded models and full checkpoints. The model gets loaded first by initializing a model with `meta` (read: empty) weights and then it determines how to load the sharded model across the available GPUs. It employs a simple pipeline parallelism method and while this is not the most efficient method, it's the most flexible for a large variety of models. See this language modeling guide for a quick glance in how this works. For instance, with `./GALACTICA/lm_gen.py` we could load BLOOM 176b model with only one GPU! It might not be the most efficient execution, but hey, it works :).

The latter argument `load_in_8bit` makes it possible to load in a model while using less memory. This approach 8-bit quantizes the model with super minimal performance loss. The main idea is to make large language models more accessible with a smaller infrastructure. For instance, this method allows us to load the full BLOOM 176b model on eight A100 40GB GPUs, as opposed to using 16 A100 GPUs. However, as nothing is free in life, this comes at the cost of inference time. We can expect forward propagation slow downs of 16-40%. I encourage you to read this blog post as it's a good read (or, the paper).

Generation

As we tokenize our input and load our model we can easily generate a piece of text given our input by using Huggingface's generate function which is implemented for CausalLMs:

```
generate_kwargs = dict(max_new_tokens=args.num_tokens, do_sample=True, temperature=args.
↪temperature)

outputs = model.generate(**input_tokens, **generate_kwargs)
```

I trust that most of these arguments are familiar to us. The `input tokens` is a dictionary containing the tokenized input text (`input_ids`), an optional `attention mask` and `token_type_ids`. For the record, `token_type_ids` is not accepted by galactica-type models. Most of the time we are only interested in the `input_ids`, but some models require the other tensors as input as well.

DeepSpeed-Inference

The script `./GALACTICA/lm_gen_ds.py` contains code to run model inference with deepspeed. The biggest difference with `./GALACTICA/lm_gen.py` is the way deepspeed has to be initialized. Luckily, for our purposes for now this can remain minimal:

```
model = deepspeed.init_inference(
        model=model,      # Transformers models
        dtype=torch.float16, # dtype of the weights (fp16)
        replace_method=None, # Lets DS autmatically identify the layer to replace
        replace_with_kernel_inject=False, # replace the model with the kernel injector
    )
```

Deepspeed deploys Tensor parallelism that mainly distributes each layer ''horizontally"; it splits up the layer and distributes it across the GPUs, each shard then lives on its appointed gpu. Additionally, it gives us the capability to replace some modules with specialized CUDA kernels to run these layers faster. I've run this but we are not getting the correct output. This should be fixable though.

We have been having OOM problems running `lm_gen` with the `deepspeed` launcher. The galactica-6.7b model and any smaller model should work without the deepspeed launcher but we are yet to fix this for models such as gpt-neox-20b or bigger. We consistently see a 2x speedup using Deepspeed. Check out this tutorial that helped us setting this up.

Deepspeed ZeRO is an add-on to the usual DeepSpeed pipeline, it also performs sharding in a tensor parallelism fashion but with, what they call, ''stage 3" it is able to do some intelligent tensor off-loading. This can come in particularly handy with large models such as BLOOM 176b or OPT-175b. We haven't been able to get this one off the grounds for reasons unknown; it seems to get stuck forever, while generating with regular deepspeed takes a few seconds.

See the following links for more information about `ZeRO stage-3`:

1. https://www.deepspeed.ai/2021/03/07/zero3-offload.html
2. https://www.deepspeed.ai/tutorials/zero/
3. https://www.deepspeed.ai/2022/09/09/zero-inference.html

2.1.2 2. How to run as a module on Snellius

To module load OptimizedLMs add the following line to your bashrc:

```
export MODULEPATH="$MODULEPATH:/projects/0/hpmlprjs/scripts
source ~/.bashrc
```

Now we can load the module you linked to in your .bashrc.

```
module load OptimizedLMs
```

And then run with

```
lm_gen model_choice input output num_tokens temperature
```

Anoter way is to load and install your own packages:

The scripts `./GALACTICA/lm_gen.py` and `./GALACTICA/lm_gen_ds.py` can be run as is with the correct dependencies.

```
module load 2021
module load Python/3.9.5-GCCcore-10.3.0
module load PyTorch/1.11.0-foss-2021a-CUDA-11.6.0
module load Miniconda3/4.9.2

pip install mpi4py, deepspeed, pydantic
pip install transformers==4.24, accelerate
```

And then run:

```
python lm_gen.py --model_path ./language_models/galactica-6.7b/ --batch_size 2␣
↪--num_tokens 1000 --input_file ./texts/inputs/geny.txt --temperature 0.95 --
↪output_file ./texts/generations/out
```

Supported Models

For now, we have briefly tested the following models with `accelerate`.

1. galactica-6.7b
2. opt-30b
3. gpt-neox-20b
4. BLOOM

The weights of these models live under `/projects/0/hpmlprjs/GALACTICA/language_models/`.

Attention: As of now, deepspeed-inference is only compatible with galactica-6.7b.

2.1.3 3. Examples

Let's run a few examples.

```
lm_gen galactica-6.7b alpha.txt out 75 0.95
```

Where `alpha.txt` contains:

```
"The function of proteins is mainly dictated by its three dimensional␣
↪structure. Evolution has played its part in"
```

Output:

The function of proteins is mainly dictated by its three dimensional structure. Evolution has played its part in selecting the best possible protein structure that can perform its functions. This structure is called native structure and it corresponds to the minimum of potential. There are several methods to compute the structure of a protein starting from amino acid sequence. With the help of evolutionary knowledge, experimental information and many other techniques like computational tools etc. we have made significant progress in prediction of

This took 5.5s to generate excluding model loading (the model fits in memory). We actually generated a batch of 4 examples in 5.5s. With `lm_gen_ds` we generate this same batch size in 2.7s! For reference, running opt-30b with `lm_gen` takes 8s.

If you feel like it, you can run `lm_gen BLOOM input out 50 0.95` and see how it takes ~40 minutes to run.

2.2 Profiling with PyTorch

[ ]:
```
# Kill old TensorBoard sessions
import os
username = os.getenv('USER')
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)
```

[ ]:
```
import os
from typing import Sequence, Tuple
from datetime import datetime

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset
import torchmetrics.functional as metrics
import numpy as np
from torchvision import datasets, transforms, models
import matplotlib.pyplot as plt

%matplotlib inline

DATA_PATH = os.getenv('TEACHER_DIR', os.getcwd()) + '/JHL_data'
os.environ["OMP_NUM_THREADS"]="3"
```

2.2.1 What is profiling?

According to wikipedia:

"Profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization, and more specifically, performance engineering."

What this means is that you analyse your program, trying to identify bottlenecks, and thereby optimizing it's execution. As an example, you might have an application needs to read a lot of input data (quite typical in machine learning!) during its run. A profile might show you that while your code runs, your processor is mostly idling since it is waiting for input data. This might give you a hint on how to optimize your program: maybe you can read in *part* of the input, and already start computing on that *while* you're loading in your next samples. Or: maybe you can copy your data to a faster disk, before you start running.

2.2.2 Why should I care about profiling?

You may know that training large models like GPT-3 takes several *million* dollars source and a few hundred MWh source. If the engineers that trained these models did *not* spend time on optimization, it might have been several million dollars and hunderds of MWh more.

Sure, the model you'd like to train is probably not quite as big. But maybe you want to train it 10000 times, because you want to do hyperparameter optimization. And even if you only train it once, it may take quite a bit of compute resources, i.e. money and energy.

2.2.3 When should I care about profiling?

Well, you should *always* care if your code runs efficiently, but there's different levels of caring.

From personal experience: if I know I'm going to run a code only once, for a few days, on a single GPU, I'll probably not create a full profile. What I *would* do is inspect my GPU and CPU utilization during my runs, just to see if it is *somewhat* efficient, and if I didn't make any obvious mistakes (e.g. accidentally *not* using the GPU, even if I have one available).

If I know that I'll run my code on multiple GPUs, for multiple days, (potentially) on multiple nodes, and/or I need to run it multiple times, I know that my resource footprint is going to be large, and it's worth spending some time and effort to optimize the code. That's when I'll create a profile. The good part is: the more often you do it, the quicker and more adapt you become at it.

2.2.4 Define the necessary functions of code

First, we will define the necessary functions to run the training. That means we define - A model (as a class deriving from nn.Module) - Some code to plot the accuracy / loss curves - Some code to set the device on which we want to execute - A train & test loop (this time with some profiling code)

Define model

[ ]:
```
class CIFAR10CNN(nn.Module):
    def __init__(self):
        super().__init__()

        # 4 convolution layers, with a non-linear activation after each.
        # maxpooling after the activations of the 2nd, 3rd, and 4th conv layers
        # 2 dense layers for classification
        # log_softmax
        #
        # As for the number of channels of each layers, try to experiment!

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, padding=1),
            nn.ReLU(),

            nn.Conv2d(in_channels=8, out_channels=32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )

        # in_features of the first layer should be the product of the output shape of␣
↪your feature extractor!
        # E.g. if the output of your feature extractor has size (batch x 128 x 4 x 4),␣
↪in_features = 128*4*4=2048
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=2048, out_features=2048),
            nn.ReLU(),
            nn.Linear(in_features=2048, out_features=10),
            nn.LogSoftmax(dim=1)
        )



    def forward(self, x):
        features = self.feature_extractor(x)

        return self.classifier(features)
```

Define function for plotting metric curves

```
[ ]: def plot_metric_curve(
         train_metric: Sequence[float],
         val_metric: Sequence[float],
         n_epochs: int,
         metric_name: str, # Label of the y-axis, e.g. 'Accuracy'
         x_axis_name: str = 'Epoch'
     ):
         # create values for the x-axis
         train_steps, val_steps = map(
             lambda metric_values: np.linspace(start=0, stop=n_epochs, num=len(metric_
     ↪values)),
             (train_metric, val_metric)
         )

         plt.plot(train_steps, train_metric, label='train')
         plt.plot(val_steps, val_metric, label='validation')
         plt.title(f"{metric_name} vs. {x_axis_name}")
         plt.legend()
         plt.show()
```

Define some code to select the right device

```
[ ]: use_cuda = torch.cuda.is_available()
     print(f"CUDA is {'' if use_cuda else 'not '}available")
     device = torch.device("cuda" if use_cuda else "cpu")

     if use_cuda:
         torch.cuda.set_per_process_memory_fraction(0.22)
```

Define train and test loop

The `train(...)` function is just for reference: this is the same train function you used in the CIFAR10 hands-on earlier. Then, we define the `train_profiling(...)` function, which is essentially the same training loop, but with the necessary code added to generate the profiles. Finally, we specify the `test(...)` function. This is the same as from the CIFAR10 hands-on earlier, but note that in theory we could also profile the testing part. Usually, training takes by far the most time, and is therefore the most important to optimize. There are exceptions however, e.g. if you use certain metrics that are very heavy to compute, and you want to compute those on a large test dataset. In that case, optimizing the test loop might make sense as well.

```
[ ]: # This train(...) loop is just for reference, so that you can compare to train_
     ↪profiling(...)
     def train(model, device, train_loader, optimizer, epoch, log_interval=10):
         model.train()
         for batch_idx, (data, target) in enumerate(train_loader):
             # move data and target to the gpu, if available and used
             data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
     ↪target))
```

(continues on next page)

(continued from previous page)

```
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()

        accuracy = metrics.accuracy(output, target)

        if batch_idx % log_interval == 0:
            print(
                f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                f'\tLoss: {loss.detach().item():.6f}'
                f'\tAccuracy: {accuracy.detach().item():.2f}'
            )

        yield loss.detach().item(), accuracy.detach().item()

# This is the actual train loop we will use for profiling
def train_profiling(model, device, train_loader, optimizer, epoch, log_interval, logdir,␣
↪break_idx=None):
    model.train()

    # Create a torch.profiler.profile object, and call it as the last part of the␣
↪training loop
    with torch.profiler.profile(
    activities=[
        torch.profiler.ProfilerActivity.CPU,
        torch.profiler.ProfilerActivity.CUDA],
    schedule=torch.profiler.schedule(
        wait=10,
        warmup=10,
        active=10),
    on_trace_ready=torch.profiler.tensorboard_trace_handler(logdir, worker_name='worker0
↪'),
    record_shapes=True,
    profile_memory=True,  # This will take 1 to 2 minutes. Setting it to False could␣
↪greatly speedup.
    with_stack=True
) as p:
        for batch_idx, (data, target) in enumerate(train_loader):
            # move data and target to the gpu, if available and used
            data, target = map(lambda tensor: tensor.to(device, non_blocking=True),␣
↪(data, target))

            optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            loss.backward()
            optimizer.step()

            accuracy = metrics.accuracy(output, target)
```

(continues on next page)

(continued from previous page)

```
            if batch_idx % log_interval == 0:
                print(
                    f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                    f'\tLoss: {loss.detach().item():.6f}'
                    f'\tAccuracy: {accuracy.detach().item():.2f}'
                )

            yield loss.detach().item(), accuracy.detach().item()

            p.step()

            # Allow to break early for the purpose of shorter profiling
            if (break_idx is not None) and (batch_idx == break_idx):
                break

# We leave the test loop unchanged. One thing to note though is that the test loop is␣
↪decorated
# with the @torch.no_grad() decorator. This tells PyTorch that it doesn't need to compute␣
↪gradients
# in the test loops, as those are not needed. This will speed up execution.
@torch.no_grad()
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0

    for data, target in test_loader:
        # move data and target to the gpu, if available and used
        data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
↪target))

        # get model output
        output = model(data)

        # calculate loss
        test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch␣
↪loss

        # get most likely class label
        pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-
↪probability

        # count the number of correct predictions
        correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100 * correct / len(test_loader.dataset)))
```

(continues on next page)

(continued from previous page)

```
    yield test_loss, correct / len(test_loader.dataset)
```

Specify the fitting function

Here, we combine the functions we defined above into a single function that will take care of the training, validation, loop over multiple epochs, and finally plot the results. Note that we call the `train_profiling` function from here, which is the one that will generate the profile.

[ ]:
```
def fit_profiling(model, optimizer, n_epochs, device, train_loader, test_loader, log_
↪interval, logdir, break_idx):

    # get the validation loss and accuracy of the untrained model
    start_val_loss, start_val_acc = tuple(test(model, device, test_loader))[0]

    # don't mind the following train/test loop logic too much, if you want to know what's␣
↪happening, let us know :)
    # normally you would pass a logger to your train/test loops and log the respective␣
↪metrics there
    (train_loss, train_acc), (val_loss, val_acc) = map(lambda arr: np.asarray(arr).
↪transpose(2,0,1), zip(*[
        (
            [*train_profiling(model, device, train_loader, optimizer, epoch, log_
↪interval, logdir, break_idx)],
            [*test(model, device, test_loader)]
        )
        for epoch in range(n_epochs)
    ]))

    # flatten the arrays
    train_loss, train_acc, val_loss, val_acc = map(np.ravel, (train_loss, train_acc, val_
↪loss, val_acc))

    # prepend the validation loss and accuracy of the untrained model
    val_loss, val_acc = (start_val_loss, *val_loss), (start_val_acc, *val_acc)

    plot_metric_curve(train_loss, val_loss, n_epochs, 'Loss')
    plot_metric_curve(train_acc, val_acc, n_epochs, 'Accuracy')
```

Defining a custom dataloader

One of the things we learn in this profiling tutorial is the importance of an efficient data I/O pipeline. We start here with a simple custom PyTorch Dataset. This Dataset can read individual `*.png` images from a directory, and can be passed a single `*.pkl` file with all the labels in a dictionary that is indexed by the filenames of the image files (so that we know which label belongs to which image).

[ ]:
```
import pickle
import glob
from torchvision.io import read_image
from PIL import Image
```

(continues on next page)

(continued from previous page)

```
class Cifar10PNGDataset(Dataset):
    def __init__(self, label_file, img_dir, transform=None, target_transform=None):
        # Load labels
        with open(label_filename, 'rb') as fo:
            self.label_dict = pickle.load(fo, encoding='bytes')
        # List filenames with png extension
        self.img_filenames = glob.glob(os.path.join(img_dir, '*.png'))
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.label_dict)

    def __getitem__(self, idx):
        # Get filename with index idx:
        img_filename = self.img_filenames[idx]
        # Read file from disk
        # image = read_image(img_filename)
        image = Image.open(img_filename)
        # Read label from label dictionary
        label = self.label_dict[os.path.basename(img_filename)]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

Let's check that our dataloader actually works. It should show an image and its corresponding label. You can change the `sample_idx` to check different samples

```
[ ]: PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA)

# Inspect one image and one label as example:
sample_idx = 0
img_sample = train_dataset[sample_idx][0]
label_sample = train_dataset[sample_idx][1]

#plt.imshow(img_sample.permute(1,2,0))
plt.imshow(img_sample)
print(f"Label: {label_sample}")
print(f"(0 = airplane, 1 = automobile, 2 = bird, 3 = cat, 4 = deer, 5 = dog, 6 = frog, 7␣
↪= horse, 8 = ship, 9 = truck)")
```

2.2.5 Creating the profile

To create the profile, we now run a single epoch of the training

[ ]:
```
BATCH_SIZE = 128
EPOCHS = 1
LEARNING_RATE = 1e-4
NUM_DATALOADER_WORKERS = 1
BREAK_AFTER_N_ITERATIONS = 60

LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

model = CIFAR10CNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalize the data to 0 mean and 1 standard deviation, now for all channels of RGB
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA,␣
↪transform=transform)

train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=True,
        num_workers=NUM_DATALOADER_WORKERS
)

test_loader = torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=False, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=False,
        num_workers=NUM_DATALOADER_WORKERS
)

logdir = "logs/baseline/" + datetime.now().strftime("%Y%m%d-%H%M%S")

fit_profiling(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_
↪INTERVAL, logdir, BREAK_AFTER_N_ITERATIONS)
```

2.2.6 Inspect the profile

Now that we have generated a profile, we want to inspect it. For that, we will start a TensorBoard session in the cell below. It will give you a link to the running tensorboard instance.

If, for some reason, you were not able to generated the logs yourself, you can uncomment the line `logdir = ...` and inspect a reference profile that we generated for you.

WARNING: profiles (so called 'trace files') contain a lot of data. It may take a while (up to a minute or so) before the TensorBoard interface actually displays the data.

[ ]:
```
%%capture --no-stdout

# REFERENCE PROFILE:
# logdir = os.path.join(DATA_PATH,"logs/baseline/ref")

import random, os

rand_port = random.randint(6000, 8000)
username = os.getenv('USER')

# Kill old TensorBoard sessions
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)

%reload_ext tensorboard
# Run with --load_fast=false, since current TensorBoard version has an issue with the␣
↪profiler plugin
# (more info https://github.com/tensorflow/tensorboard/issues/4784)
%tensorboard --logdir=$logdir --port=$rand_port --load_fast=false

print(f"Go to https://jupyter.lisa.surfsara.nl/jhlsrf016/user/{username}/proxy/{rand_
↪port}/#pytorch_profiler")
```

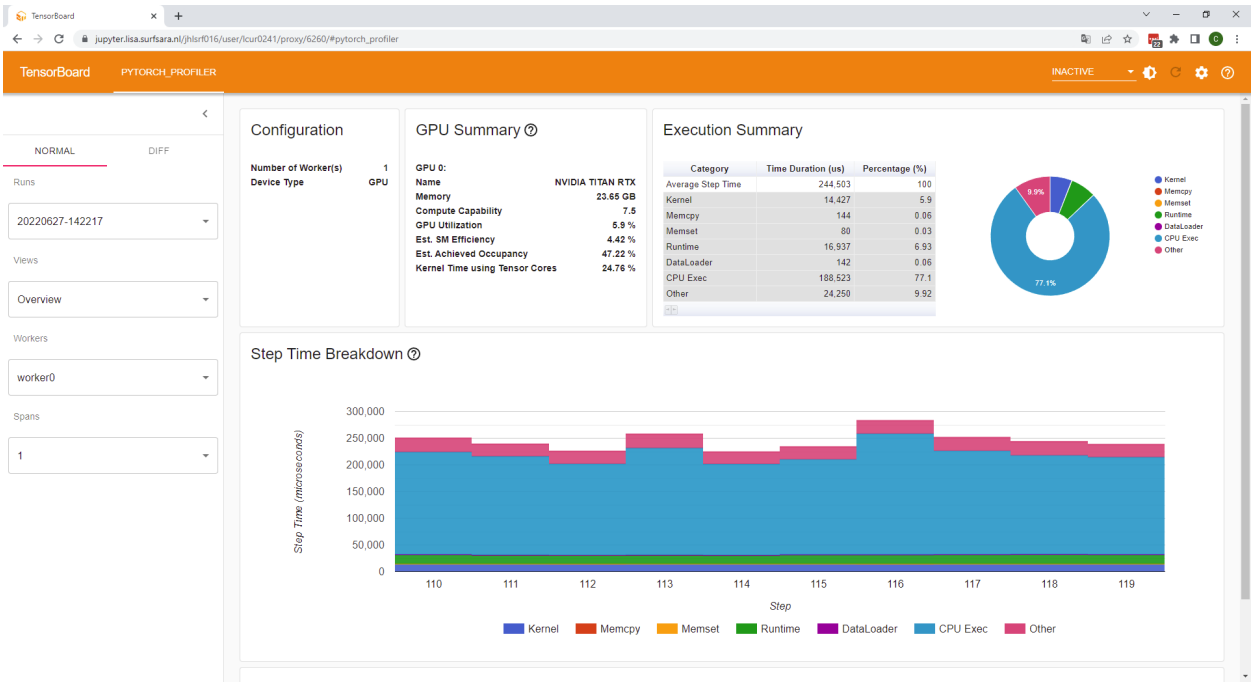
2.2.7 Understanding the profiler output

Views

In the first fiew, you see an overview of the profiling. This is probably the display that will provide the most actionable insights. What do we see? This blog contains a very detailed explaination. Below, we will cover the basics, but if you are going to profile your own code, the blog may prove very useful to get the most out of the profiler output.

Step Time Breakdown

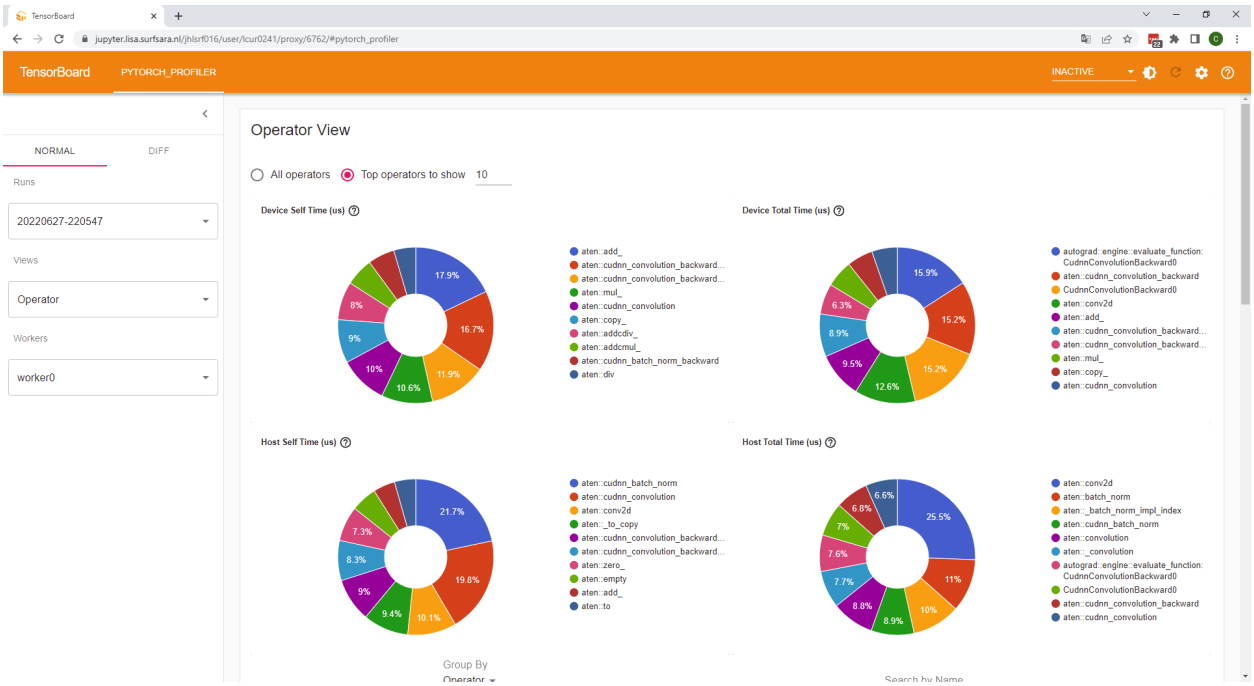
Let's start with the central part, the Step Time Breakdown.



Here, we see how long each 'Step' took, and what that time was spent on. A 'Step' is a single training step, i.e. a forward and backward pass on a single batch. If you look carefully at the `train_profiling(...)` code, and in particular the `torch.profiler.profile` call we did there, you see that we passed `wait=10, warmup=10, active=10` as arguments. What that means is, it waits for the first 10 steps (i.e. step 0-9), then, the profiler gets activated, but discards it results (for step 10-19), and then 10 steps get recorded (i.e. step 20-29). This cycle will keep repeating itself as long as we were training. With a batch size of 128 and 50000 images in total, the cycle would normally be repeated 13 times if we trained for one epoch. The PyTorch profiler calls each of these cycles a 'Span'. In this case however, we cut the training shorter for faster profiling, and only trained for 60 iterations. Thus, we'll only see two spans in our profile. Thus, this cycle repeated 13 times.

Spans

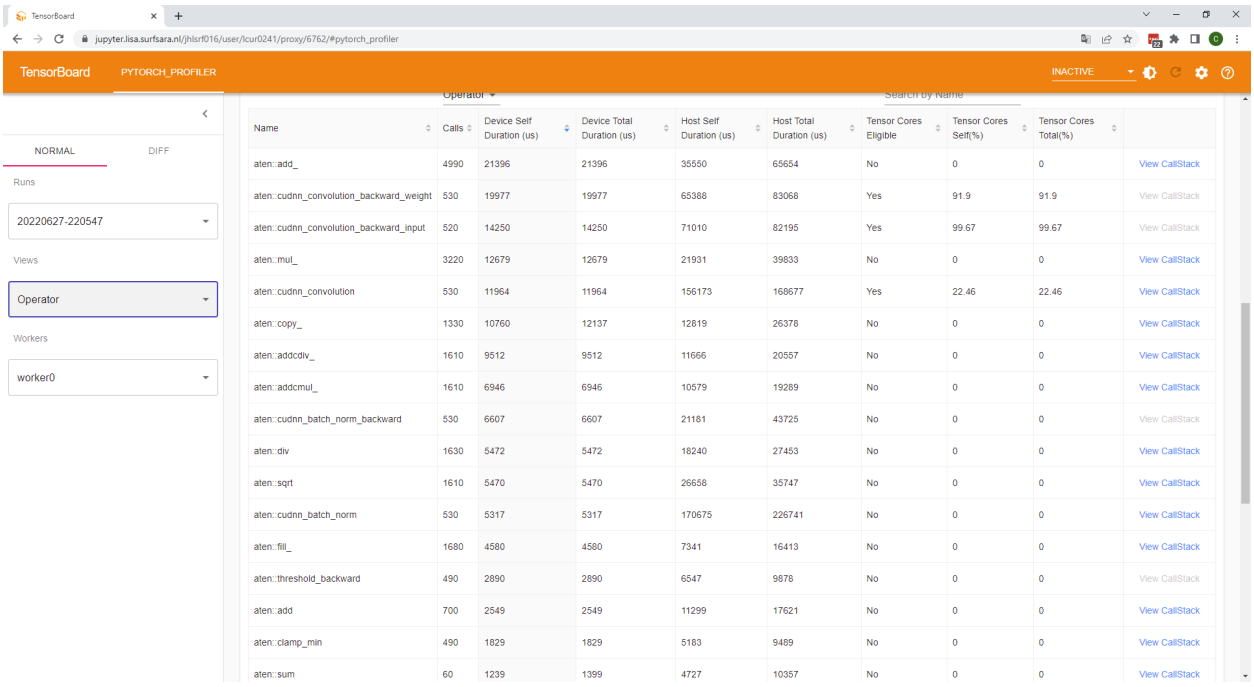
Click the 'Spans' button. Here, you see that indeed, we recorded 2 'spans'. If you click on span 2, you'll see that this contains the timing for step 50-59.



Typically, recording only a few iterations/spans is enough. In very particular cases, you might be interested how the speed develops over time. Then, it is useful to record and inspect multiple spans.

GPU Summary

The next window that is useful (if training on GPUs) is the GPU Summary. However, in order to understand the GPU summary, it's good to know a little bit about how cores on a GPU are organized. Below, you see the schematic layout of an Nvidia A100 GPU chip (source)

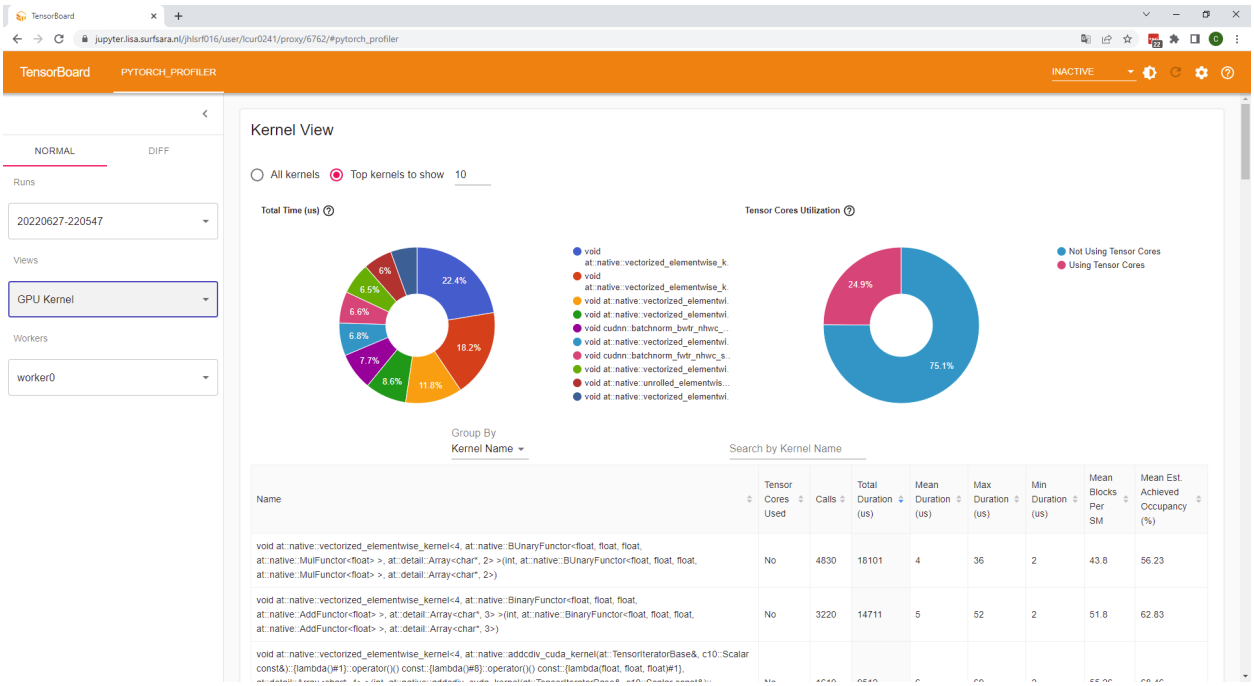


The tiny green dots in this image are GPU cores. They are organized together in groups, so called SM (or: streaming

multiprocessor) units. An A100 GPU has 128 SM units. Below, we see a schematic representation of a single SM on an Nvidia A100 GPU

Here, you see that a single SM has many different cores: 16 cores that can perform integer operations, 16 cores that can perform single precision floating point operations (i.e. operations on numbers represented in memory by 32 bits), 8 cores that can perform double precision floating point operations (i.e. operations on numbers represented in memory by 64 bits) and a block of Tensor Cores (special cores for tensor operations). An SM only has a single instruction unit. What that means that all cores in an SM can only perform *one* operation at the same time. I.e. we can not have 1 FP32 core doing subtraction, and the other multiplication, within the same SM unit. Of course, *different* SM units can work on different things. I.e. this particular GPU could in theory run 128 *different* operations, and run each of those operations on e.g. 16 single precision floating point numbers *in parallel*. This massive parallelism is what makes a GPU so good at processing many data elements with the same operation, as is done in a lot of matrix operations.

Ok, back to the summary.



The first thing to note here is the GPU Utilization. As we can see, in this example it is only 3.39%. That means that 96.61% of the time our GPU was doing *nothing*, typically because it is waiting for other resources (I/O, CPU, ...). That's bad: we have a very expensive GPU in this machine, and we're hardly using it!

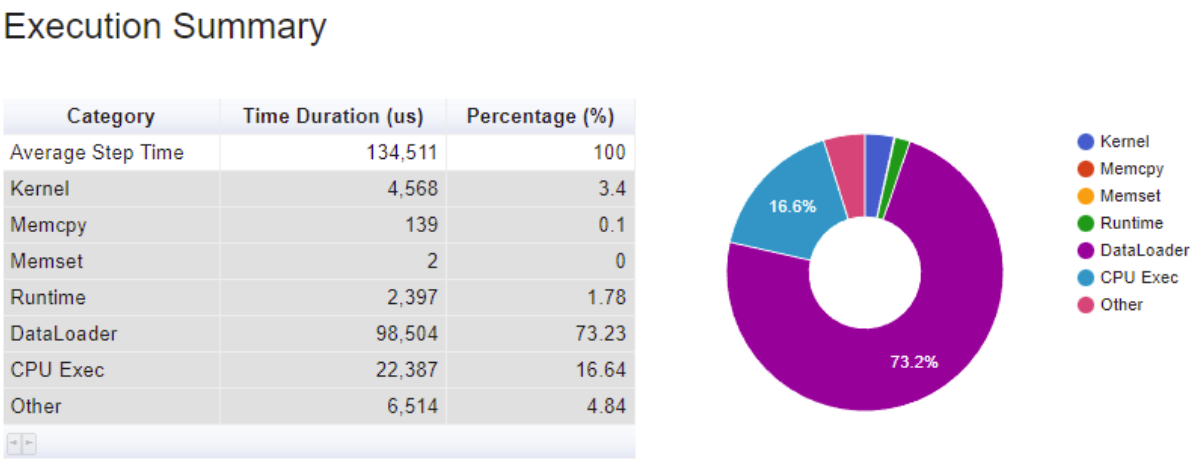
Note that a high GPU utilization does not *necessarily* mean we are using the GPU efficiently. Even if just a *single* core (out of the thousands that a GPU has) is working on a task the full time, GPU utilization would be 100%. However, it would mean we are only using a single core, on a single SM, leaving all of those others idle. As you can see, a high GPU utilization is not a *guarantee* for efficient usage, but it is a *minimum requirement*.

The Est SM Efficiency provides us some deeper insight. It is the (average) fraction of SMs that were in use over the profiled time period. A low percentage here means SMs have been idling. The aforementioned extreme case of a single core (in a single SM) working all the time would immediately be identified with this metric: GPU utilization would be 100%, but Est SM Efficiency would be 1/128 (in case of an A100 with 128 SM units in total). However, a high Est SM Efficiency is *still* not enough to guarantee efficient usage. A use case that occupies *one* core in each SM would also show a high Est SM Efficiency. In a machine learning workload, (partially) empty SMs can occur if there are operations being performed on relatively small matrices. That can be because input sizes are small, or because the batch size is small. I.e. batch size is *one* of the things you can explore if you see a low Est. SM Efficiency.

Finally, we see Kernel Time using Tensor Cores. We'll get back to those later.

Execution Summary

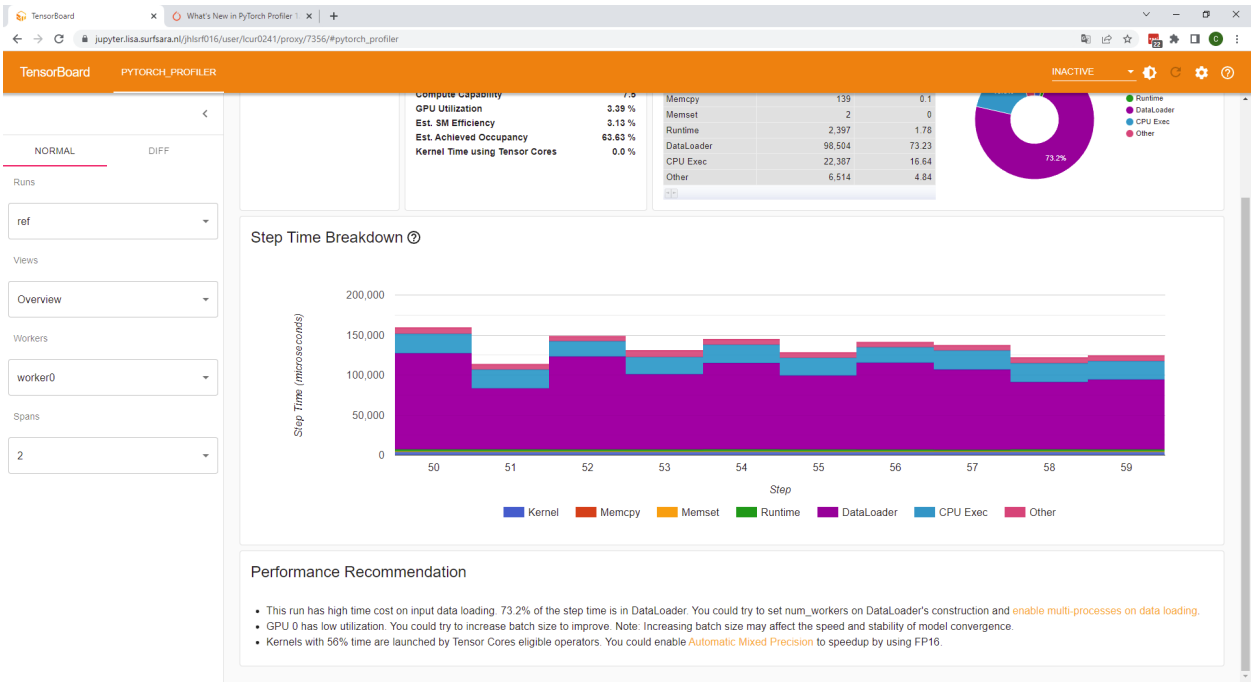
The execution summary gives is an overview of which components took the most time (averaged over the steps in this Span).



Here, we see that the average step time was 134 ms. The vast majority of that (98ms) was taken by data loading. Another substantial chunk was CPU execution.

Performance Recommendation

The final part of the overview window is the performance recommendation. Based on the timings measured in the profile, the PyTorch Profiler will try to give you some advice on what to do to speed up your code.



Often, these tips are quite decent, though the Profiler doesn't know what you've already done. Also, which piece of advice will help the most might be better judged by you.

2.2.8 Improving I/O performance

Increasing the number of workers for dataloading

We've seen that 73.2% of our time was spent in dataloading, rather than computing anything. Clearly, this is the first problem we want to tackle. Let's start by following the PyTorch profilers advice, and increase the amount of workers (i.e. CPU cores) used for dataloading.

First, let's see how many cores we have available in this environment:

[ ]:
```
def get_cpu_count():
    return len(os.sched_getaffinity(0))

print(f"Number of CPUs: {get_cpu_count()}")
```

Since the bulk of our computation happens on the GPU, we can probably use all of the CPU cores we have available for dataloading (if you would be training on the CPUs as well, it might be more efficient to designate a few CPU cores for dataloading, and the rest for training).

Try setting the NUM_DATALOADER_WORKERS to 3,

[ ]:
```
BATCH_SIZE = 128
EPOCHS = 1
LEARNING_RATE = 1e-4
NUM_DATALOADER_WORKERS = 3
BREAK_AFTER_N_ITERATIONS = 60

LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

model = CIFAR10CNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalize the data to 0 mean and 1 standard deviation, now for all channels of RGB
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA,␣
↪transform=transform)

train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=True,
        num_workers=NUM_DATALOADER_WORKERS
)

test_loader = torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=False, transform=transform, download=True),
```

(continues on next page)

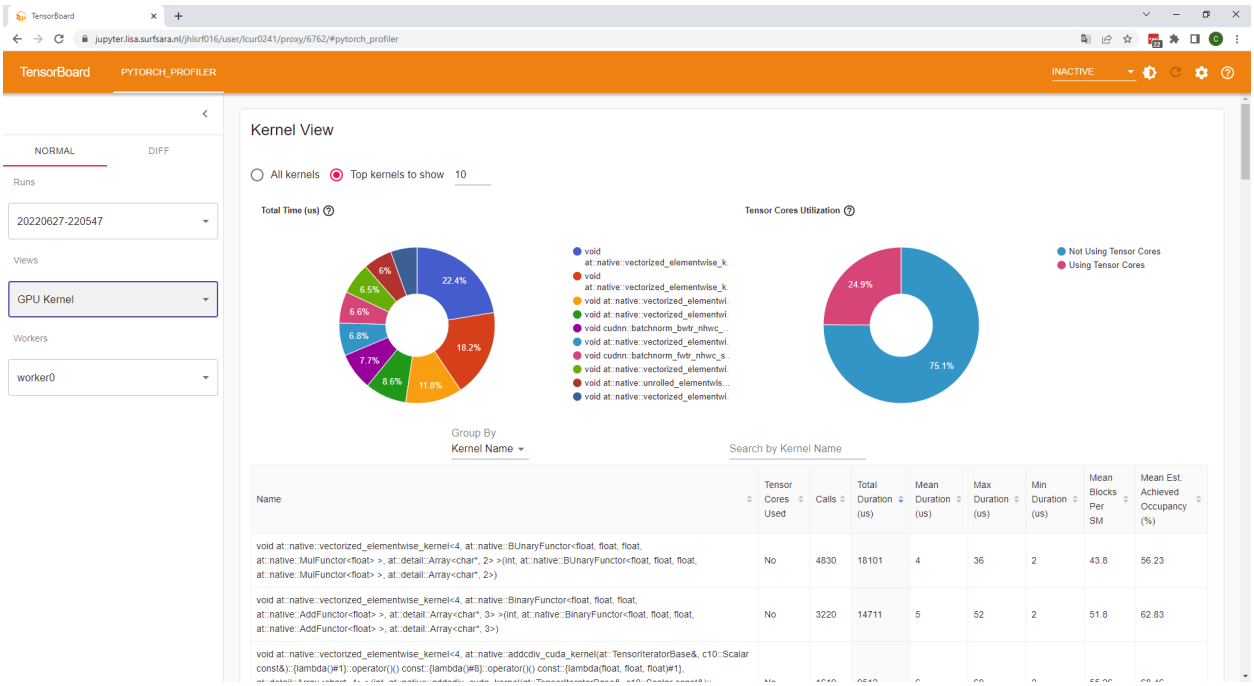
(continued from previous page)

```
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=False,
        num_workers=NUM_DATALOADER_WORKERS
)

logdir = "logs/num_dataloaders/" + datetime.now().strftime("%Y%m%d-%H%M%S")

fit_profiling(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_
↪INTERVAL, logdir, BREAK_AFTER_N_ITERATIONS)
```

[ ]:
```
%%capture --no-stdout

# REFERENCE PROFILE:
# logdir = os.path.join(DATA_PATH,"logs/num_dataloaders/ref")

import random, os

rand_port = random.randint(6000, 8000)
username = os.getenv('USER')

# Kill old TensorBoard sessions
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)

%reload_ext tensorboard
# Run with --load_fast=false, since current TensorBoard version has an issue with the␣
↪profiler plugin
# (more info https://github.com/tensorflow/tensorboard/issues/4784)
%tensorboard --logdir=$logdir --port=$rand_port --load_fast=false

print(f"Go to https://jupyter.lisa.surfsara.nl/jhlsrf016/user/{username}/proxy/{rand_
↪port}/#pytorch_profiler")
```

Inspecting & Interpreting results

You should see something like this:



As we can see, the average step time has gone down substantially, to 134 to 39 ms. Note that for codes that are limited by I/O, you might see large variations: we are working on a network filesystem here, meaning that all of us in the course are reading from the same disks. If one of your fellow students is hitting the filesystem at the same time as you are, it might slow down substantially. This variation is also visible between step 51, 54 and 57.

Note that this variation is ***not*** the reason we see so little dataloader time at e.g. step 52 and 53. The data for step 51-53 gets loaded during step 48-50 by the three dataloaders. However, since the computation is faster than the dataloading, the 3 dataloaders are not yet finished when step 51 starts. Thus, step 51 is waiting for the dataloaders to finish before it can start computing, which is what you see as 'DataLoader' time in the profile. Once that I/O is completed, we immediately have ***three*** batches, since we had three dataloaders. That's why step 52 and 53 don't show any Dataloading time anymore.

As we can see, GPU utilization has also gone up to 11% or so. Not great, but still a lot better than before.

Transformations

Transformations are also part of the Dataloading pipeline. However, these are performed on the CPU and that may take substantially more time that the GPU needs to subsequently train the network. One way to speedup the dataIO is to perform the normalization as part of the network itself, so that it can be executed on the GPU. Note that this is not *always* useful: it depends on the balance between the amount of work between the GPU and the CPU. If the dataloading on the CPU can already keep up with the GPU, there's no point in offloading more operations to the GPU. In this case, we have a pretty light network however, and we see if we can shave off a little bit more of the dataloading time.

Below, we redefine the network, with the first layer now being the normalization.

[ ]:
```
class CIFAR10CNN(nn.Module):
    def __init__(self):
        super().__init__()

        # 4 convolution layers, with a non-linear activation after each.
        # maxpooling after the activations of the 2nd, 3rd, and 4th conv layers
        # 2 dense layers for classification
```

(continues on next page)

(continued from previous page)

```
        # log_softmax
        #
        # As for the number of channels of each layers, try to experiment!

        self.feature_extractor = nn.Sequential(
            transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
            nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, padding=1),
            nn.ReLU(),

            nn.Conv2d(in_channels=8, out_channels=32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )

        # in_features of the first layer should be the product of the output shape of␣
↪your feature extractor!
        # E.g. if the output of your feature extractor has size (batch x 128 x 4 x 4),␣
↪in_features = 128*4*4=2048
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=2048, out_features=2048),
            nn.ReLU(),
            nn.Linear(in_features=2048, out_features=10),
            nn.LogSoftmax(dim=1)
        )


    def forward(self, x):
        features = self.feature_extractor(x)

        return self.classifier(features)
```

[ ]:
```
BATCH_SIZE = 128
EPOCHS = 1
LEARNING_RATE = 1e-4
NUM_DATALOADER_WORKERS = 3
BREAK_AFTER_N_ITERATIONS = 60

LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

model = CIFAR10CNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)
```

(continues on next page)

(continued from previous page)

```
transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalization now done in the network
    # transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])


PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA,␣
↪transform=transform)

train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=True,
        num_workers=NUM_DATALOADER_WORKERS
)

test_loader = torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=False, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=False,
        num_workers=NUM_DATALOADER_WORKERS
)

logdir = "logs/transforms/" + datetime.now().strftime("%Y%m%d-%H%M%S")

fit_profiling(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_
↪INTERVAL, logdir, BREAK_AFTER_N_ITERATIONS)
```

[ ]:
```
%%capture --no-stdout

# REFERENCE PROFILE:
# logdir = os.path.join(DATA_PATH,"logs/transforms/ref")

import random, os

rand_port = random.randint(6000, 8000)
username = os.getenv('USER')

# Kill old TensorBoard sessions
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)

%reload_ext tensorboard
# Run with --load_fast=false, since current TensorBoard version has an issue with the␣
↪profiler plugin
# (more info https://github.com/tensorflow/tensorboard/issues/4784)
%tensorboard --logdir=$logdir --port=$rand_port --load_fast=false
```

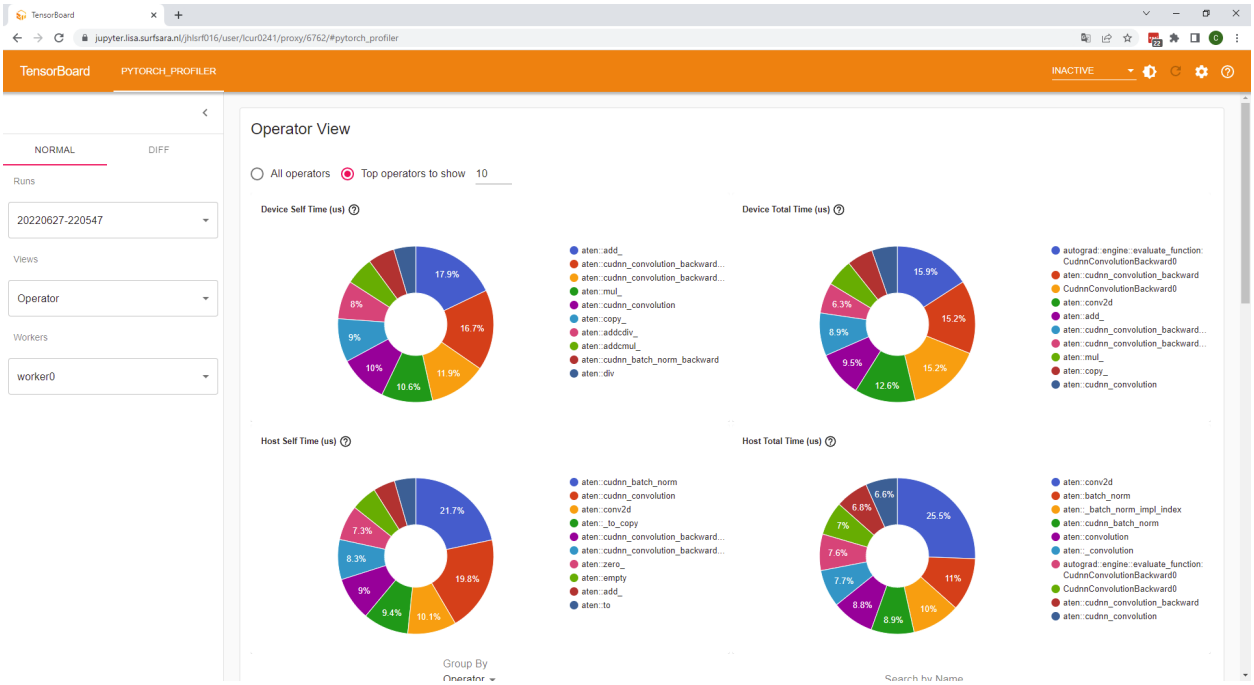
(continues on next page)

(continued from previous page)

```
print(f"Go to https://jupyter.lisa.surfsara.nl/jhlsrf016/user/{username}/proxy/{rand_
↪port}/#pytorch_profiler")
```

Inspecting and interpreting results

You should see something like this



We've managed to reduce step time to 33s now: not a huge gain, but the dataloading part is only 1.9s per step now on average. So, there's not much more to gain. You may see a peak at some steps. If you're eager, you can try to increase the `prefetch_factor`, which is an additional argument you can pass to the `torch.utils.data.DataLoader`. This means more samples will be loaded in advance, and you're less sensitive to fluctuations in I/O speed. However, it will increase (CPU) memory consumption, as you'll need to hold more samples in (CPU) memory. If you feel up to it, you can add it in the code above, rerun the profiling and start a new TensorBoard.

Final remarks on optimizing I/O

In this case, we've managed to reduce the data loading overhead to almost zero. The reason is that the dataset we work on in this tutorial is tiny: 198 MB in total. This is simply cached by our network filesystem, and therefore I/O is very fast.

In general however, the way this dataset is stored, as one file per sample (image), is ***very bad*** for performance. This is because filesystems need to read metadata (where is the file located, how big is it, updating the last access date, etc) for every individual file. On average, each cifar10 image is about 2KB. Imagine how large the overhead of reading overhead (metadata) for such a small file! This is not a huge issue on e.g. a local laptop with an SSD, since since you're the only one using that disk and since SSDs can do metadata operations very quickly. On large clusters however, metadata operations are (comparatively speaking) very slow. Large clusters typically use network filesystems that are optimized for throughput, i.e. bandwidth, but can process relatively modest amounts of files per second.

Thus, generally, it is advised to pack samples together in a packed file format. It doesn't hurt on your local laptop, and is all-but-essential on large clusters. Many packed file formats are available, examples are LMDB, HDF5/h5py, TFRecords, petastorm, or even zip or tarballs (but don't compress, since uncompressing takes time too!). One thing to note is that you'll want a packed file format where you can read a *single* sample without having to load the whole file in memory.

In some clusters, the nodes have local storage. In that case, an option to increase the performance is to first copy the data to local storage. But: also these data copies are a lot faster if you copy a single file, rather than a large amount of small files. So even in those cases, packed file formats are benificial.

2.2.9 New baseline: a heavier model

Of course, in this tutorial, we've been mostly playing with a toy model. Realistic models are a lot heavier to train. Let's try a real RESNET50 model. Torchvision has such a model defined by default. Let's use it to set a new baseline for our training run.

[ ]:
```
model = models.resnet50()
print(model)
```

Because this code produces much larger trace files, we decrease the number of active iterations, and stop the training after 16 iterations (i.e. one 'span'). Otherwise, the trace files will be very slow to analyze.

[ ]:
```
# This is the actual train loop we will use for profiling
def train_profiling(model, device, train_loader, optimizer, epoch, log_interval, logdir,␣
↪break_idx):
    model.train()

    # Create a torch.profiler.profile object, and call it as the last part of the␣
↪training loop
    with torch.profiler.profile(
    activities=[
        torch.profiler.ProfilerActivity.CPU,
        torch.profiler.ProfilerActivity.CUDA],
    schedule=torch.profiler.schedule(
        wait=10,
        warmup=3,
        active=3),
    on_trace_ready=torch.profiler.tensorboard_trace_handler(logdir, worker_name='worker0
↪'),
    record_shapes=True,
    profile_memory=True,  # This will take 1 to 2 minutes. Setting it to False could␣
↪greatly speedup.
    with_stack=True
) as p:
        for batch_idx, (data, target) in enumerate(train_loader):
            # move data and target to the gpu, if available and used
            data, target = map(lambda tensor: tensor.to(device, non_blocking=True),␣
↪(data, target))

            optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            loss.backward()
```

(continues on next page)

(continued from previous page)

```
            optimizer.step()

            accuracy = metrics.accuracy(output, target)

            if batch_idx % log_interval == 0:
                print(
                    f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                    f'\tLoss: {loss.detach().item():.6f}'
                    f'\tAccuracy: {accuracy.detach().item():.2f}'
                )

            yield loss.detach().item(), accuracy.detach().item()

            p.step()

            # Allow to break early for the purpose of shorter profiling
            if (break_idx is not None) and (batch_idx == break_idx):
                break
```

[ ]:
```
BATCH_SIZE = 128
EPOCHS = 1
LEARNING_RATE = 1e-4
NUM_DATALOADER_WORKERS = 3
BREAK_AFTER_N_ITERATIONS = 16

LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

# Now use resnet50
model = models.resnet50()
model = model.to(device, memory_format=torch.channels_last)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalization now done in the network
    # transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])


PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA,␣
↪transform=transform)

train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=True,
        num_workers=NUM_DATALOADER_WORKERS
```

(continues on next page)

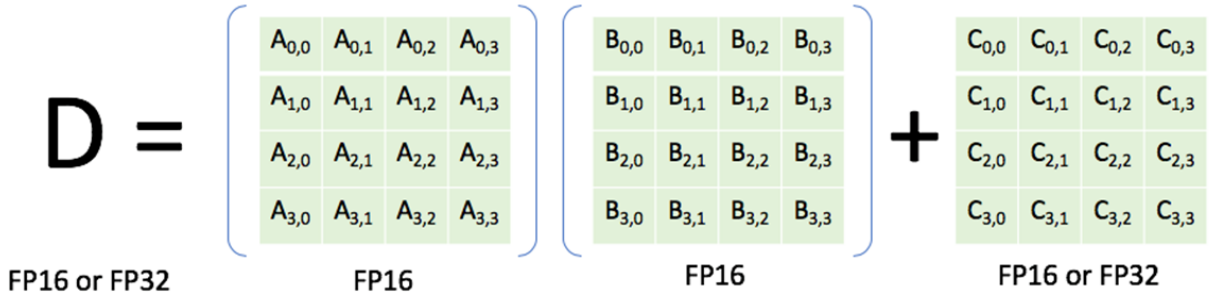
(continued from previous page)

```
)

test_loader = torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=False, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=False,
        num_workers=NUM_DATALOADER_WORKERS
)

logdir = "logs/resnet50_baseline/" + datetime.now().strftime("%Y%m%d-%H%M%S")

fit_profiling(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_
↪INTERVAL, logdir, BREAK_AFTER_N_ITERATIONS)
```

[ ]:
```
%%capture --no-stdout

# REFERENCE PROFILE:
# logdir = os.path.join(DATA_PATH,"logs/resnet50_baseline/ref")

import random, os

rand_port = random.randint(6000, 8000)
username = os.getenv('USER')

# Kill old TensorBoard sessions
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)

%reload_ext tensorboard
# Run with --load_fast=false, since current TensorBoard version has an issue with the␣
↪profiler plugin
# (more info https://github.com/tensorflow/tensorboard/issues/4784)
%tensorboard --logdir=$logdir --port=$rand_port --load_fast=false

print(f"Go to https://jupyter.lisa.surfsara.nl/jhlsrf016/user/{username}/proxy/{rand_
↪port}/#pytorch_profiler")
```

Inspecting and interpreting results

Let's have a look at our new baseline:

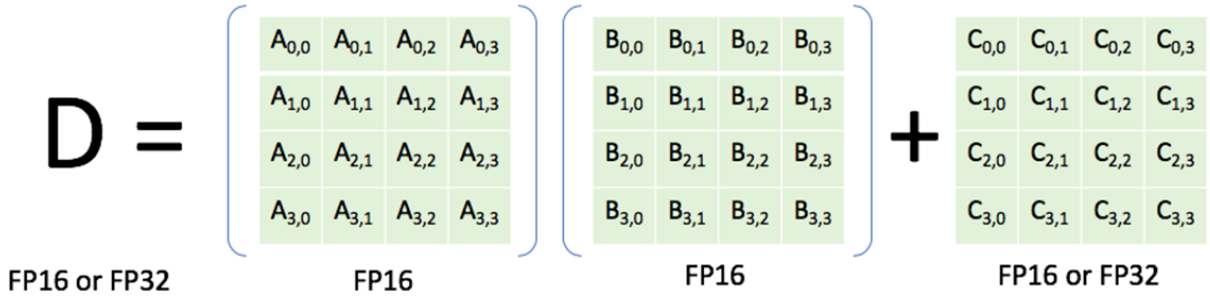


As we can see, a step now takes 229 ms, out of which 37s are spend on GPU kernel execution.

Improving GPU kernel execution time by mixed precision

Cores on an Nvidia GPU have a dedicated processing unit for processing tensor operations, so-called "Tensor Cores". These can do operations on tensors much faster than when those tensors would have to be performed with the traditional FP32 units in the core.

The operation that these Tensor Cores perform is called a fused-multiply-add, and they do it on a (small) matrix. They can do one of these operations every clock cycle of the GPU:



There are some requirements though. On the GPUs that we are working on in this training, the tensor cores only support operations where tensors A and B are stored as 16-bit precision floating point numbers. Tensor C and D can be either 16 or 32 bit. Torch has a specific feature called "automatic mixed precision" (AMP for short), which is supported through the `torch.cuda.amp` package (see this documentation). In principle, this can be used to automatically convert the right tensors in the model to 16-bit floating point, so that the Tensor Cores can be used. Note however that you may need to 'scale' your gradients: FP16 floating point numbers have a smaller representable range (since they have fewer bits to represent the number), which may cause underflows in your gradients (i.e. numbers in your gradient may become smaller than the smallest number that can be represented with the FP16 datatype). If that happens, you need to do gradient scaling using the `torch.cuda.amp.GradScaler`. In the below example, we exlcuded that, since here we just want to demonstrate the potential speedup.

[ ]:
```
from torch import autocast

# This is the actual train loop we will use for profiling
def train_profiling(model, device, train_loader, optimizer, epoch, log_interval, logdir,␣
↪break_idx=None):
    model.train()

    # Create a torch.profiler.profile object, and call it as the last part of the␣
↪training loop
```

(continues on next page)

(continued from previous page)

```
    with torch.profiler.profile(
    activities=[
        torch.profiler.ProfilerActivity.CPU,
        torch.profiler.ProfilerActivity.CUDA],
    schedule=torch.profiler.schedule(
        wait=10,
        warmup=3,
        active=3),
    on_trace_ready=torch.profiler.tensorboard_trace_handler(logdir, worker_name='worker0
↪'),
    record_shapes=True,
    profile_memory=True,  # This will take 1 to 2 minutes. Setting it to False could␣
↪greatly speedup.
    with_stack=True
) as p:
        for batch_idx, (data, target) in enumerate(train_loader):
            # move data and target to the gpu, if available and used
            data = data.to(device, non_blocking=True, memory_format=torch.channels_last)
            target = target.to(device, non_blocking=True)

            optimizer.zero_grad()

            # Here, we use automatic mixed precision:
            with autocast(device_type='cuda'):
                output = model(data)
                loss = F.nll_loss(output, target)

            loss.backward()
            optimizer.step()

            accuracy = metrics.accuracy(output, target)

            if batch_idx % log_interval == 0:
                print(
                    f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                    f'\tLoss: {loss.detach().item():.6f}'
                    f'\tAccuracy: {accuracy.detach().item():.2f}'
                )

            yield loss.detach().item(), accuracy.detach().item()

            p.step()

            # Allow to break early for the purpose of shorter profiling
            if (break_idx is not None) and (batch_idx == break_idx):
                break
```

[ ]:
```
BATCH_SIZE = 128
EPOCHS = 1
LEARNING_RATE = 1e-4
NUM_DATALOADER_WORKERS = 3
```

(continues on next page)

(continued from previous page)

```
BREAK_AFTER_N_ITERATIONS = 16

LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

# Now use resnet50
model = models.resnet50()
model = model.to(device, memory_format=torch.channels_last)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalization now done in the network
    # transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])


PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA,␣
↪transform=transform)

train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=True,
        num_workers=NUM_DATALOADER_WORKERS
)

test_loader = torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=False, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=False,
        num_workers=NUM_DATALOADER_WORKERS
)

logdir = "logs/autocast/" + datetime.now().strftime("%Y%m%d-%H%M%S")

fit_profiling(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_
↪INTERVAL, logdir, BREAK_AFTER_N_ITERATIONS)
```

[ ]:
```
%%capture --no-stdout

# REFERENCE PROFILE:
# logdir = os.path.join(DATA_PATH,"logs/autocast/ref")

import random, os

rand_port = random.randint(6000, 8000)
username = os.getenv('USER')
```

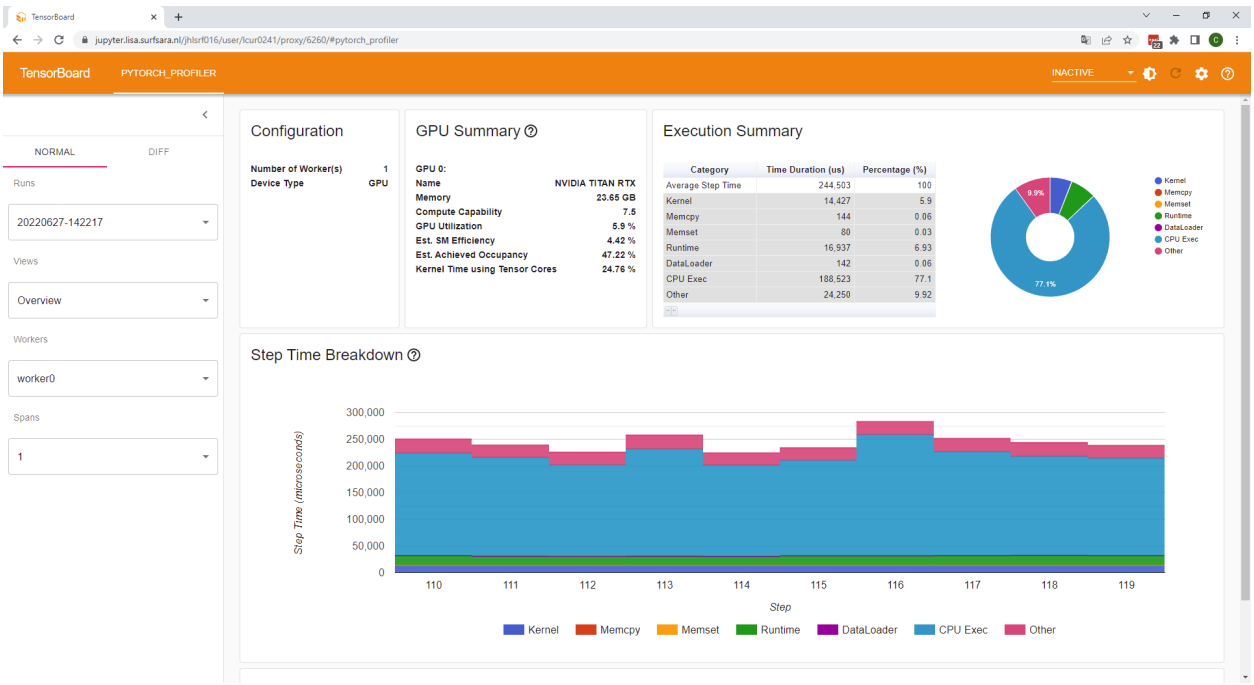
(continues on next page)

(continued from previous page)

```
# Kill old TensorBoard sessions
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)

%reload_ext tensorboard
# Run with --load_fast=false, since current TensorBoard version has an issue with the␣
↪profiler plugin
# (more info https://github.com/tensorflow/tensorboard/issues/4784)
%tensorboard --logdir=$logdir --port=$rand_port --load_fast=false

print(f"Go to https://jupyter.lisa.surfsara.nl/jhlsrf016/user/{username}/proxy/{rand_
↪port}/#pytorch_profiler")
```

Inspecting and interpreting results

You should see something like this

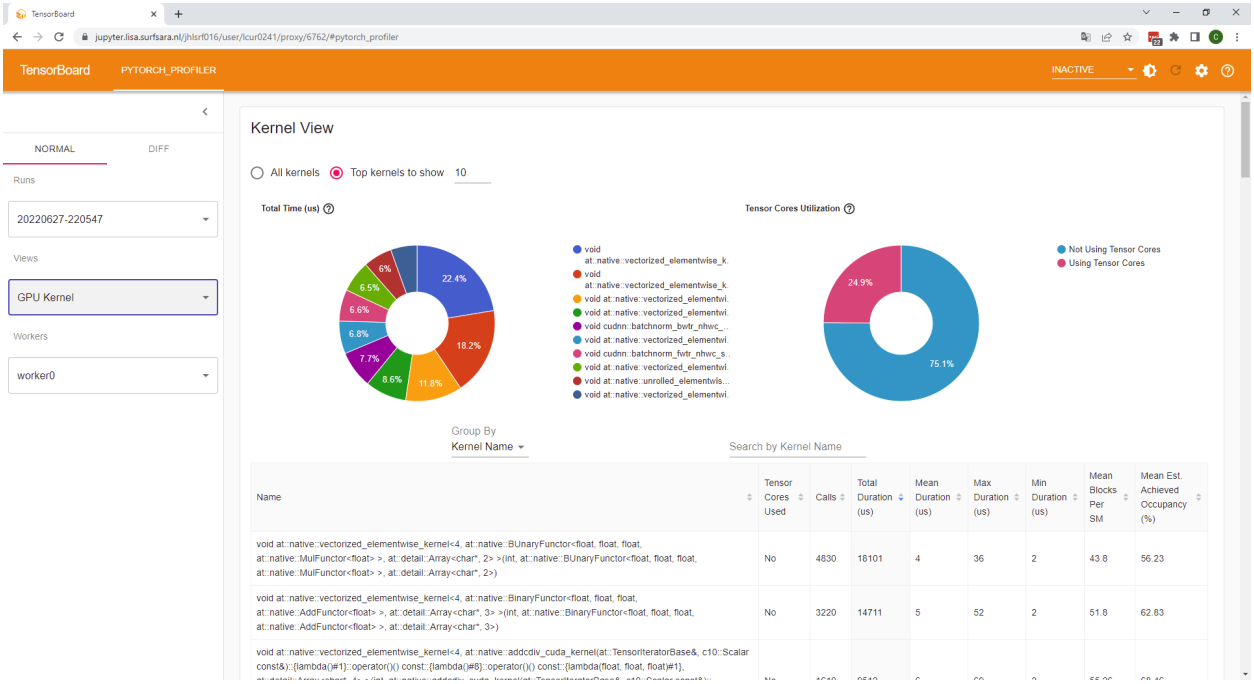


We'll focus on the GPU kernel execution time here. It has decreased from about 37 to 14 ms, and the GPU summary now tells us that almost 25% of the time spent on executing kernels, it is using the Tensor Cores. You might have noticed that GPU utilization has gone *down* in this case, compared to our baseline run. However, here, that's not a bad thing: we have essentially reduced the *amount of work* that needed to be done by moving to reduced precision.

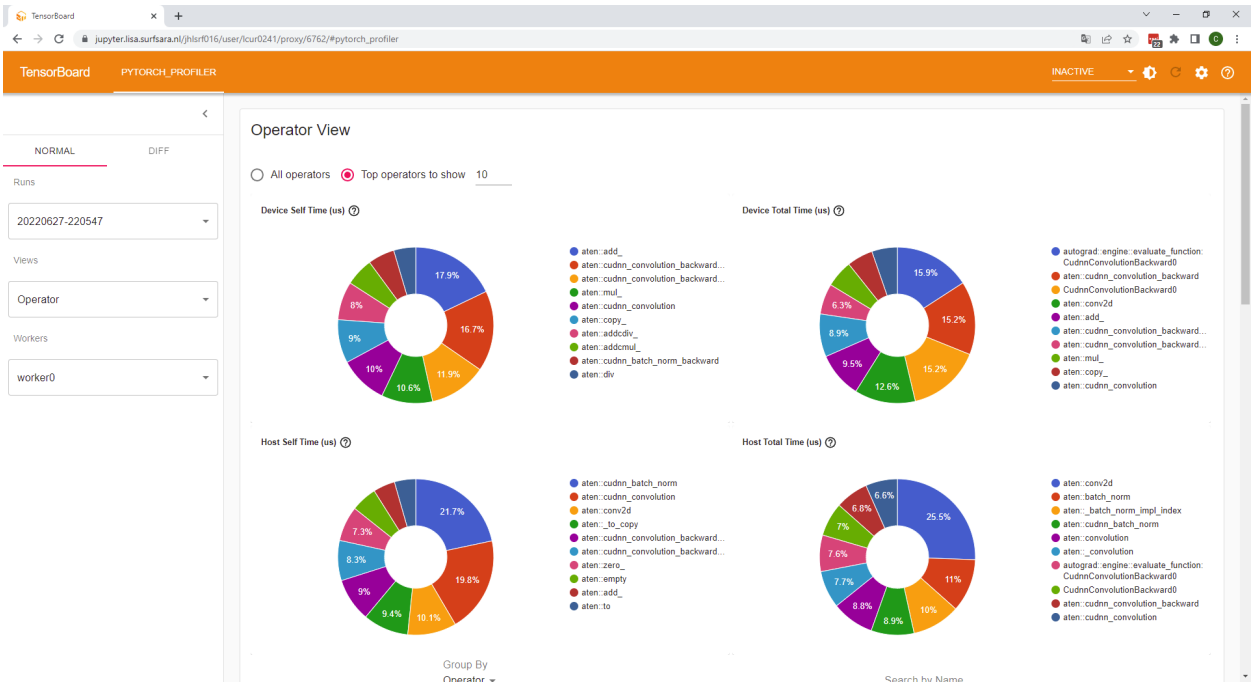
What *is* problematic though is that the CPU Exec time has gone up. While we haven't confirmed this, we think it is due to the extra overhead introduced by the type conversion from FP32 to FP16. So, for this particular network and (small) input, it is not worth it to use mixed precision - but in many real world cases it does provide a substantial speedup!

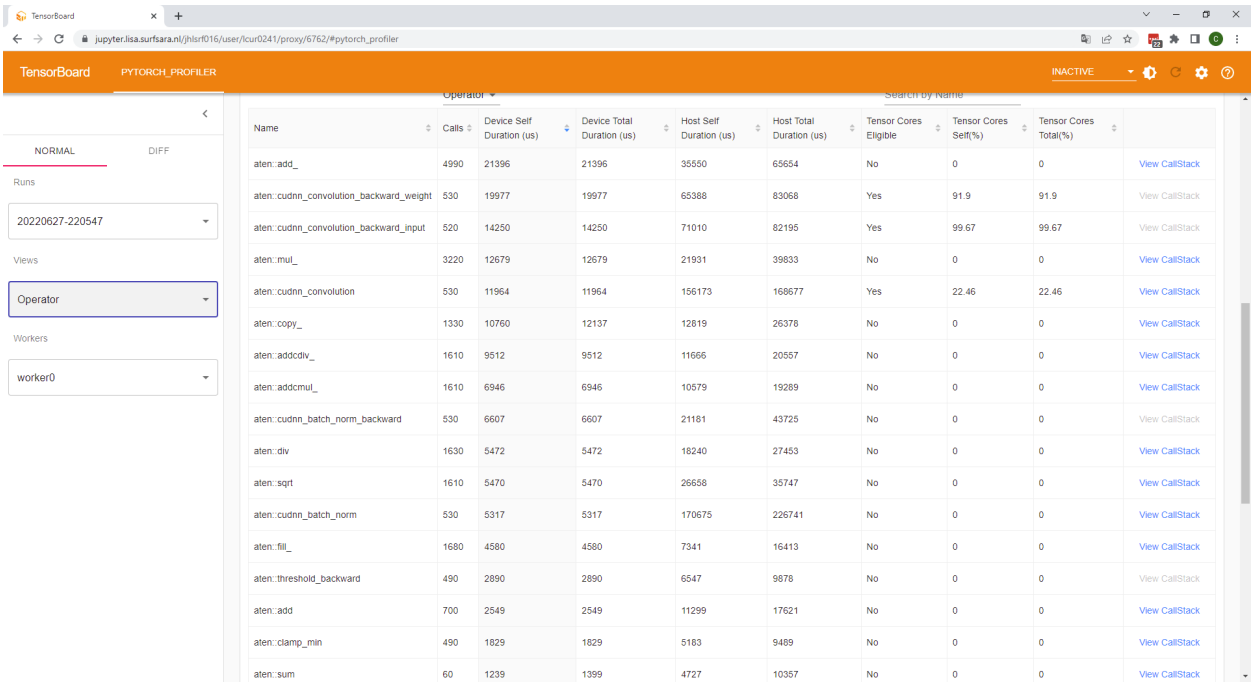
2.2.10 Other views of the profiler

Here, we'll just discuss some of the other views of the profiler, without exercises.



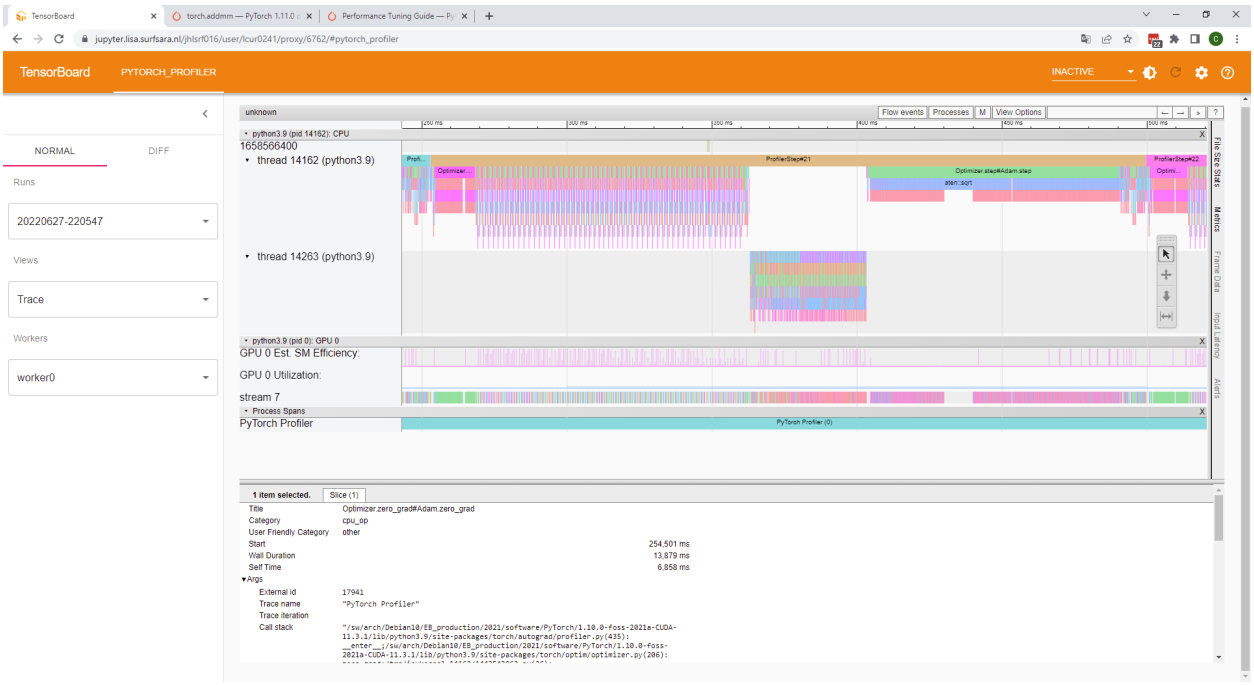
In the GPU kernel view, you see a list of all GPU kernels that have been executed. You can see a lot of details, like how often these particular kernels have been run, how much time that took, if the kernel ran on tensor cores or not, and things like mean estimated achieved occupancy (we have talked about this metric before when discussing the GPU summary). It's hard to gain actionable information from this view.



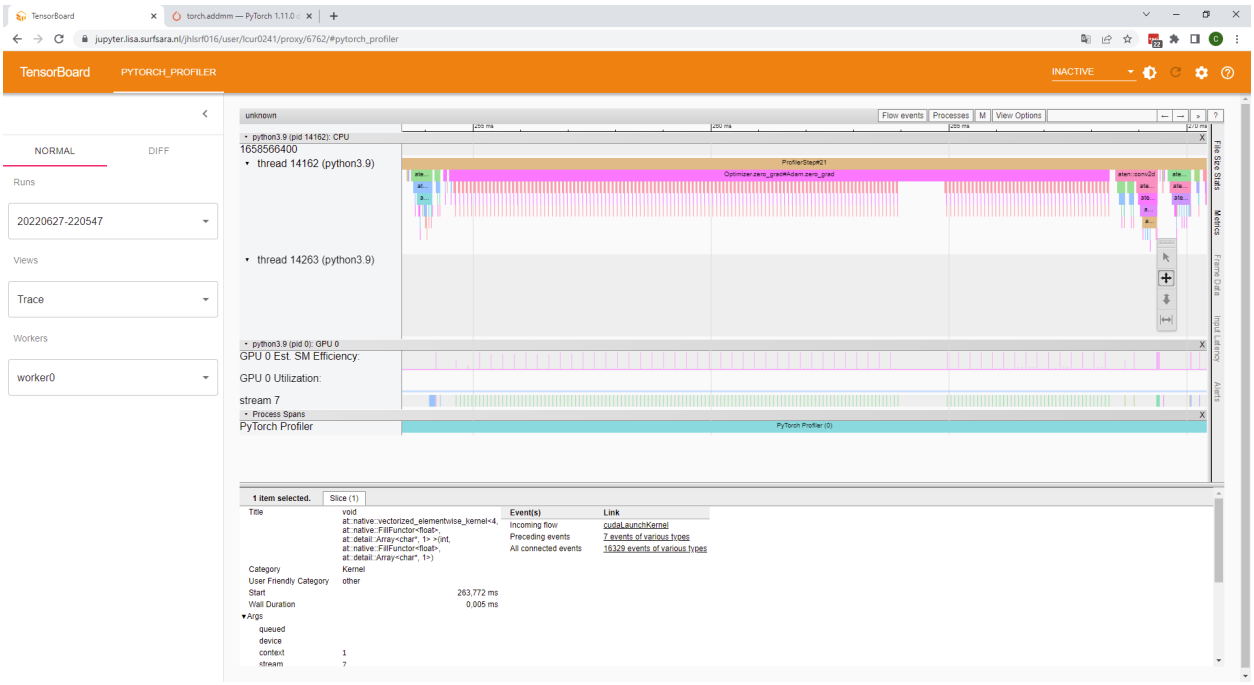


The operator view shows all PyTorch operations that have been executed. The question marks next to the pie graphs give some additional explaination about what you see there. Similarly, though it gives some interesting insights into which operators take the most time in your computation, its hard to gain actionable insight from this.

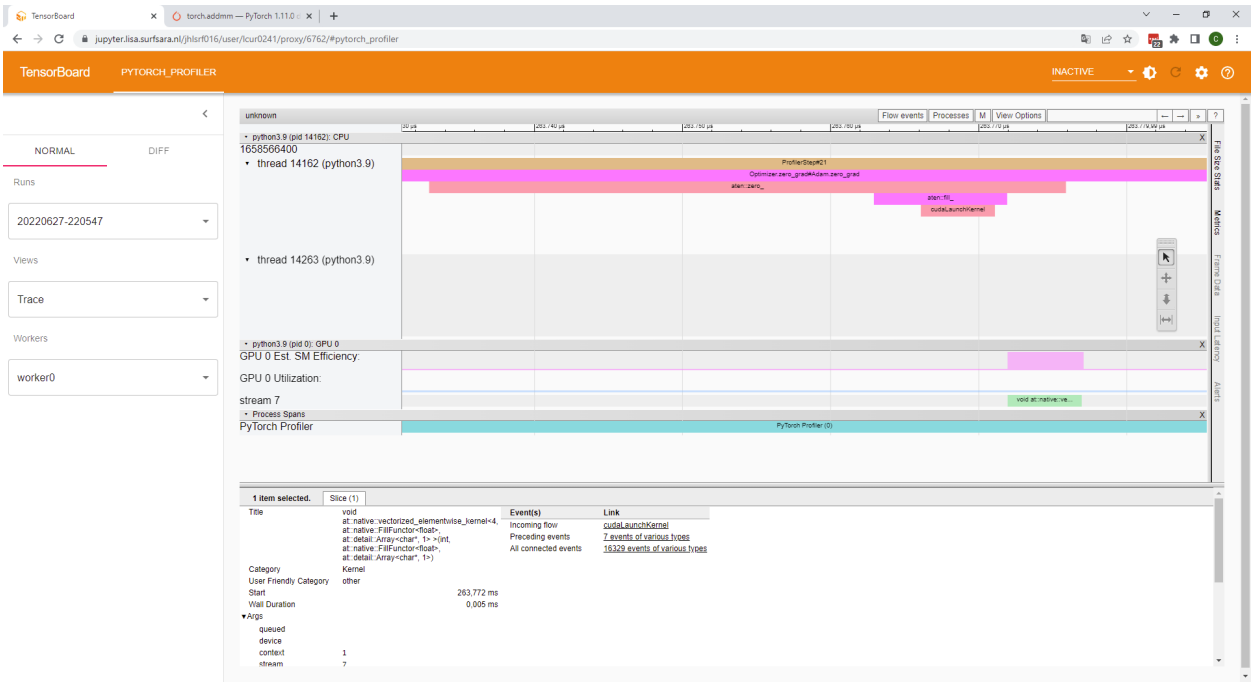
In the trace view, we see a timeline of the execution of the training iterations. At the top, we clearly recognize the 10 iterations that we profiled. Let's zoom in on one step:



The two 'threads' at the top show the processes that execute on the CPU. The 'stream 7' shows the kernels that execute on the GPU. You can click on certain operations to see what they are, and how long they took. For example, the purple block at the top left that says 'Optimizer…' is now selected. It shows that the full title is 'Optmizer.zero_grad', i.e. this is the process that fills the gradients with zeroes. Let's zoom in on that:



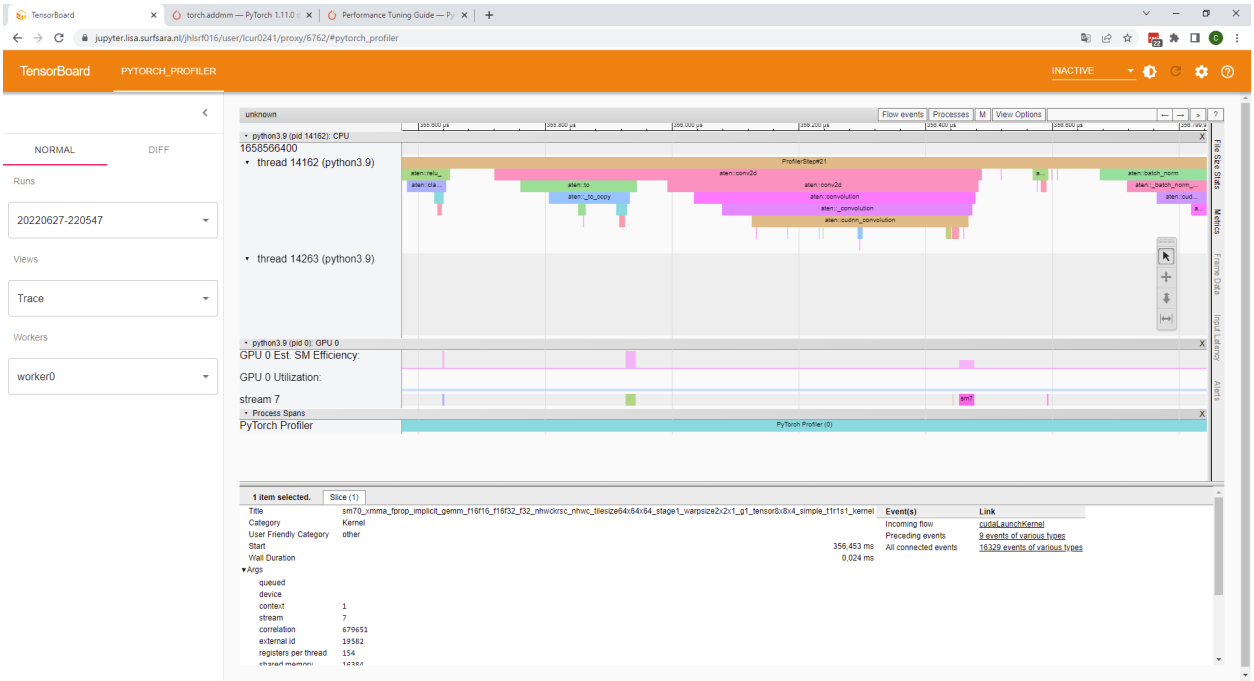
We see some pattern being repeated. Zooming in further, we see:



Essentially, you see a call stack here. `Optmizer.zero_grad` called `aten::zero_`, which called `aten::fill_`, which called `cudaLaunchKernel`, which in turn launced a cuda kernel (the green block in stream 7).

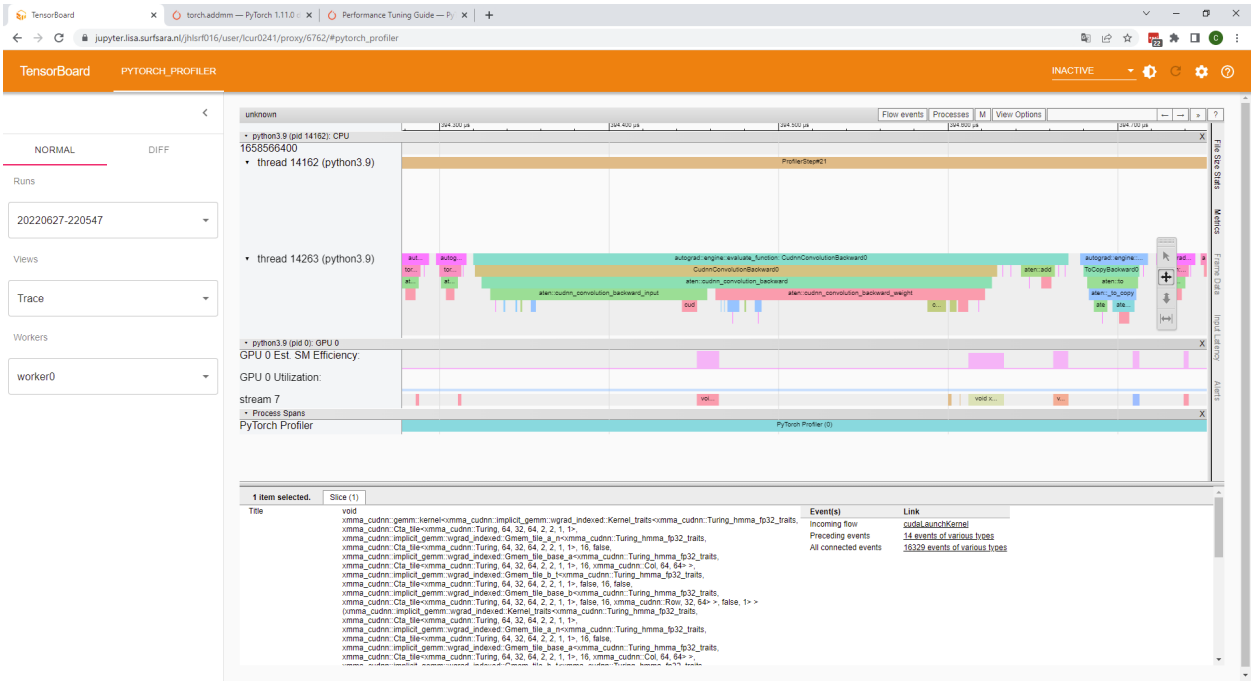
While this process doesn't take a *ton* of time, it is mentioned in the PyTorch tuning guide as one of the things that could be optimized: https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html . It is mentioned that one can set the parameters to 'None' instead of to zero explicitely, which may be more efficient, but might incur slightly different numerical behaviour.

Zooming in on another part of the step, we see e.g. the 2D convolutions from the forward pass:

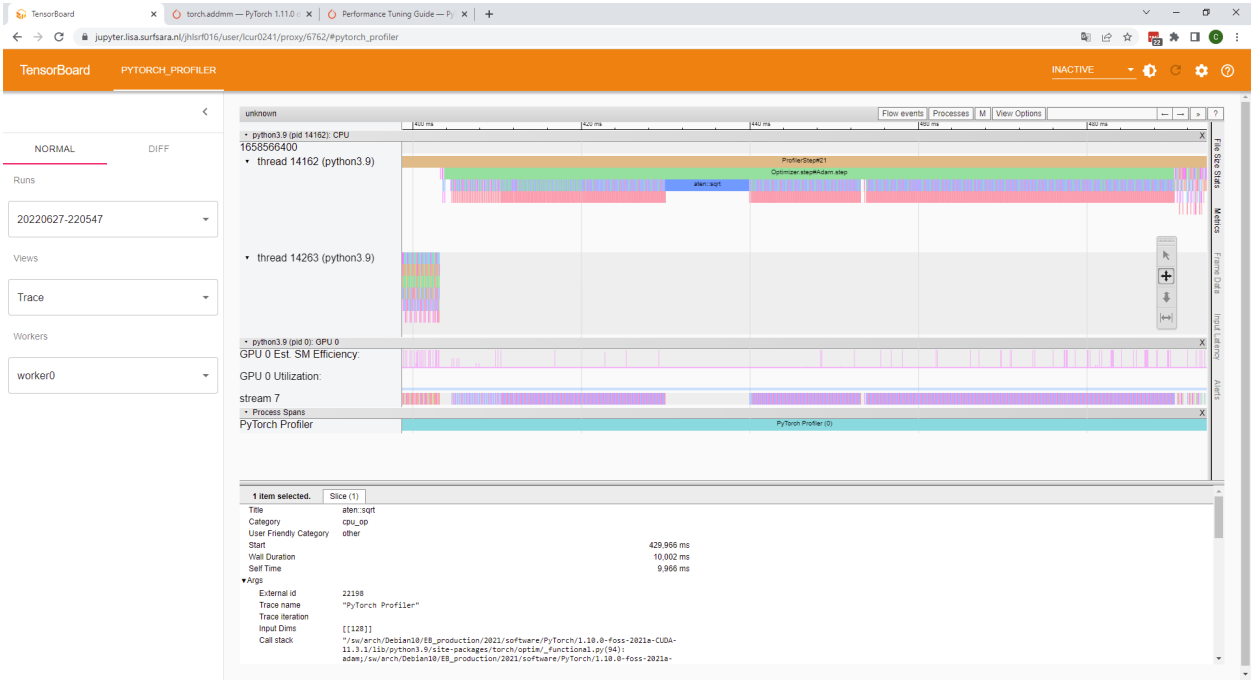


Like before, you can recognize the CPU call stack for the aten::conv2d operation, which eventually results in the laucnh of a kernel on the GPU.

Zooming in on another part, we see some typical part of the backward pass:



Finally, there's the optimizer step, which will do things like updating the weights:




```
[1]: import torch
     import torch.nn as nn
     import numpy as np
     import matplotlib.pyplot as plt
     import random
```

(continues on next page)

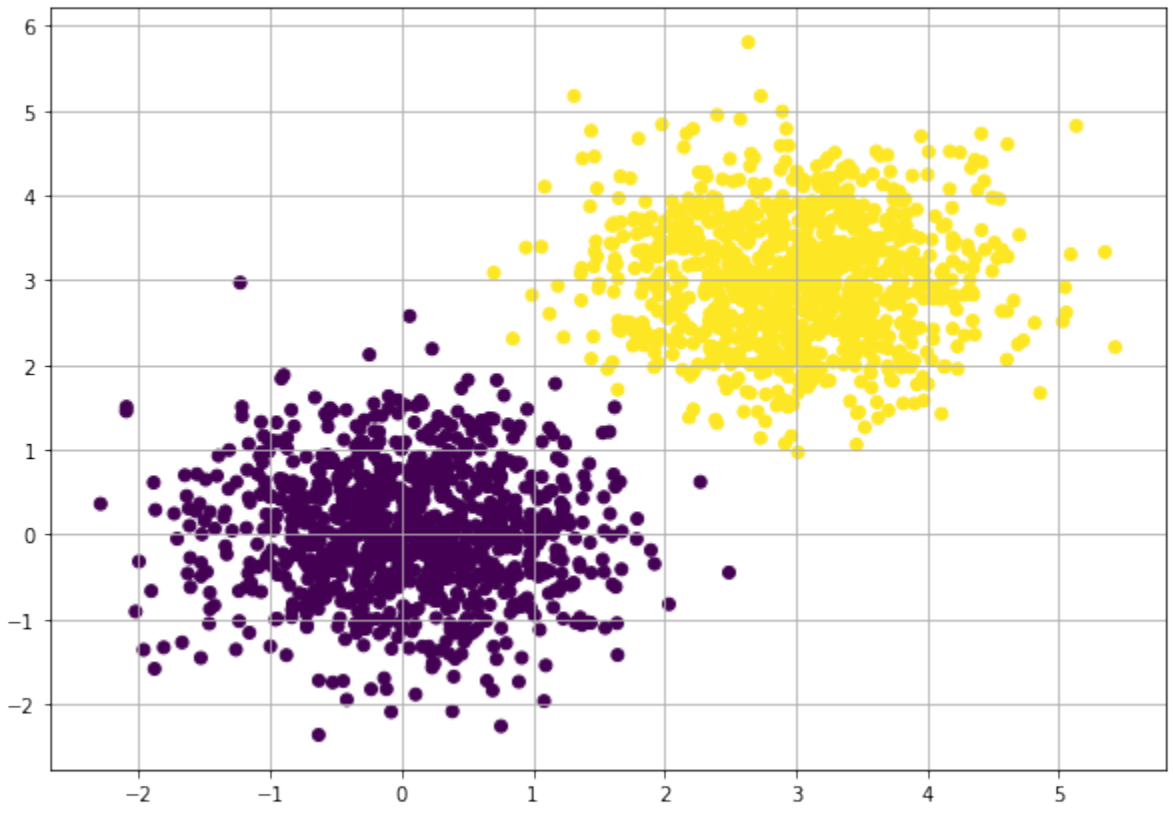
(continued from previous page)

```
# Needed to use matplotlib in a Jupyter environment
%matplotlib inline

# seeded random state for reproducibility
np.random.seed(1)
```

2.3 Generate some dummy data

[2]:
```
num_samples = 1000
# Data of class A
A = np.random.normal(0, 0.75, size=(num_samples, 2))
# Data of class B
B = np.random.normal(3, 0.75, size=(num_samples, 2))
# A is labelled as 0 and B is labelled as 1
y = [0 if i < num_samples else 1 for i in range(num_samples*2)]
input_data = np.concatenate([A, B], axis=0)
```

[3]:
```
plt.figure(figsize = (10,7))
plt.scatter(input_data[:,0], input_data[:,1], c=y)
plt.grid()
plt.show()
```

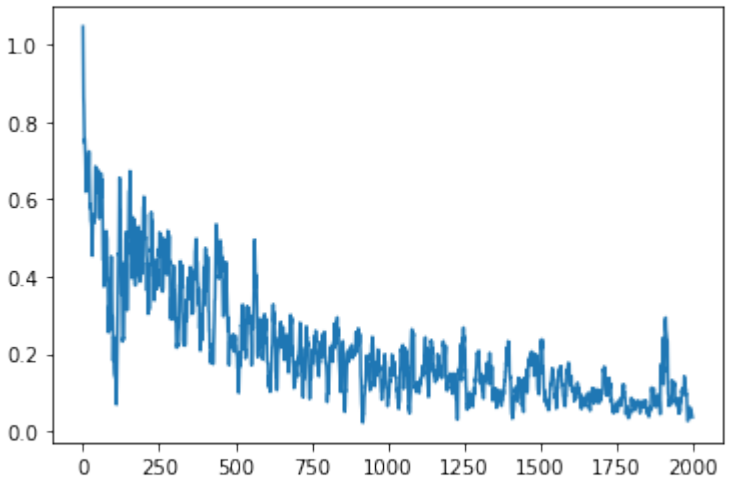


2.4 Define a neural network with PyTorch

[4]:
```
class NeuralNetwork(nn.Module):
    def __init__(self, input_dim=2, hidden_dim=8):
        super().__init__()

        self.layers = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        prediction = self.layers(x)
        return prediction
```

2.5 Train Loop

[5]:
```
data = list(zip(input_data, y))
random.shuffle(data)
model = NeuralNetwork()
model.train()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
bce = nn.BCELoss()

loss_list = []
for x, target in data:

    x, target = torch.tensor(x).unsqueeze(0).float(), torch.tensor([target]).float()
    optimizer.zero_grad()
    pred = model(x).squeeze(1)
    loss = bce(pred, target)
    loss.backward()
    optimizer.step()

    loss_list.append(loss.detach().item())

def smooth_loss_curve(loss_curve, window_size=10):
    return np.convolve(loss_curve, np.ones(window_size)/window_size)[window_size-1:]

loss_curve = smooth_loss_curve(loss_list)

plt.plot(loss_curve)
plt.show()
```



[ ]:
```
import os
from typing import Sequence, Tuple

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
import torchmetrics.functional as metrics

%matplotlib inline

DATA_PATH = os.getenv('TEACHER_DIR', os.getcwd()) + '/JHL_data'
```

2.6 Introduction to Deep Learning with PyTorch

To help you understand the fundamentals of deep learning, this demo will walk through the basic steps of building a toy model for classifying handwritten numbers with accuracies surpassing 95%. This model will be a basic fully-connected neural network.

2.6.1 The Task for the Neural Network

Our goal is to construct and train an artificial neural network on thousands of images of handwritten digits so that it may successfully identify others when presented. The data that will be incorporated is the MNIST database which contains 60,000 images for training and 10,000 test images.

Loading Training Data

The MNIST dataset is conveniently bundled within Torch Vision, and we can easily take a look at some of its features.

2.6.2 Lets take a look at our input data

[ ]:
```
example_trainset = datasets.MNIST(root=DATA_PATH, train=True, transform=transforms.
↪ToTensor(), download=True)

example_image, example_label = next(iter(example_trainset))

print(f"Input data shape: {example_image.shape}")
example_image = example_image[0] # removing first dimension

print(f"Input label: {example_label}")

plt.imshow(example_image, cmap='gray')
plt.show()
```

2.6.3 Let's plot the numerical representation of the image so that we can see what the model sees as input!

[ ]:
```
fig = plt.figure(figsize = (12,12))
ax = fig.add_subplot(111)
ax.imshow(example_image, cmap='gray')
width, height = example_image.shape
thresh = example_image.max()/2.5
for x in range(width):
    for y in range(height):
```

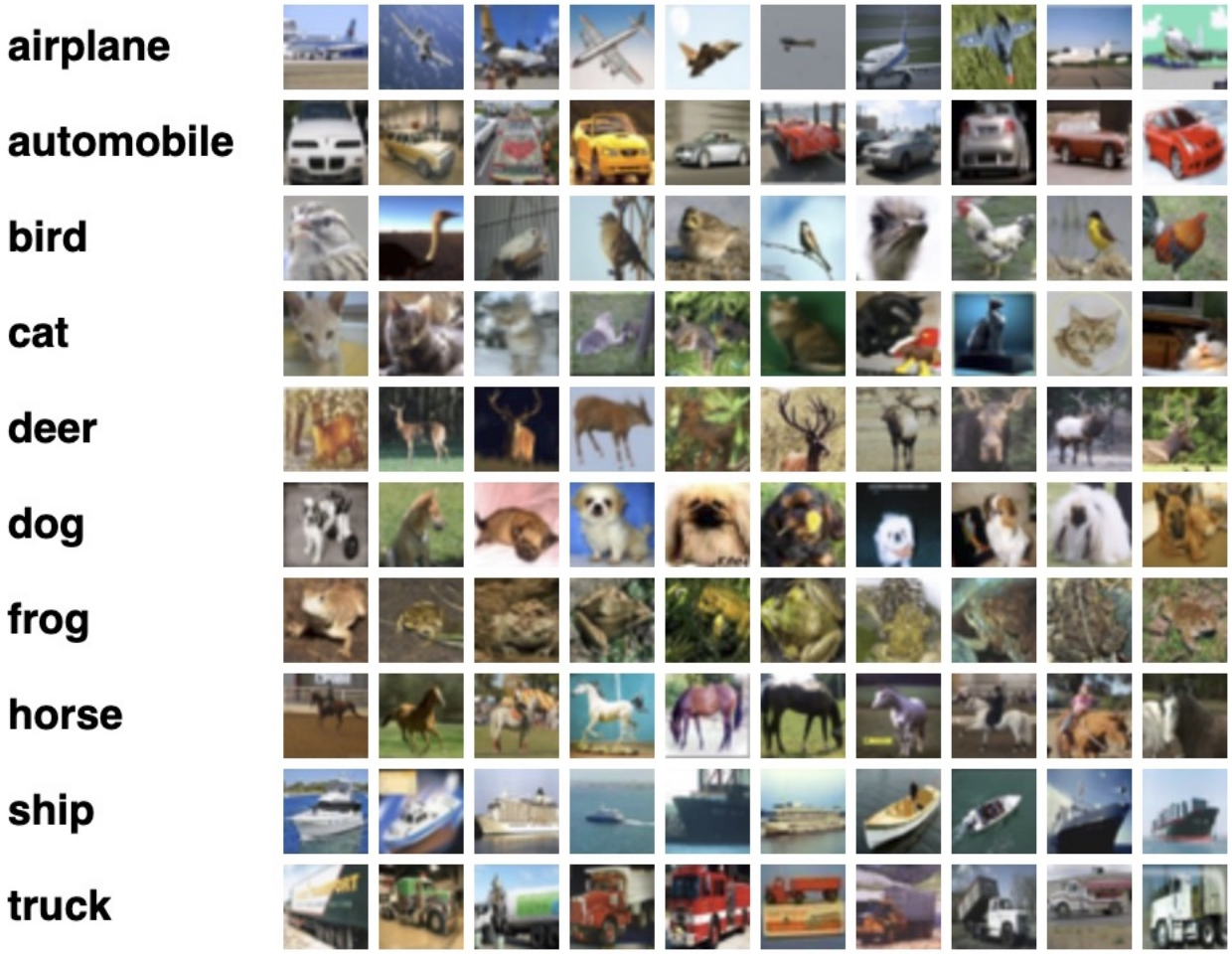
(continues on next page)

(continued from previous page)

```
        val = round(example_image[x][y].item(), 2) if example_image[x][y] != 0 else 0
        ax.annotate(str(val), xy=(y, x),
                    horizontalalignment='center',
                    verticalalignment='center',
                    color='white' if example_image[x][y]<thresh else 'black')
```

Formatting the input data layer Instead of a 28 x 28 matrix, we build our network to accept a 784-length vector.

Each image needs to be then reshaped (or flattened) into a vector.



This will be the first layer to the model we are going to build.

[ ]:
```
class LinearClassifier(nn.Module):
    def __init__(self, input_size=28*28):
        super().__init__()  # Python magic which initialises all relevant PyTorch␣
↪properties

        # Defining the model:
        # - A layer that flattens the input
        # - A fully connected layer which has 512 neurons who each connect to the full␣
↪28x28 input
        # - A fully connected layer which has 10 neurons, each of which represent an␣
↪output class,
```

(continues on next page)

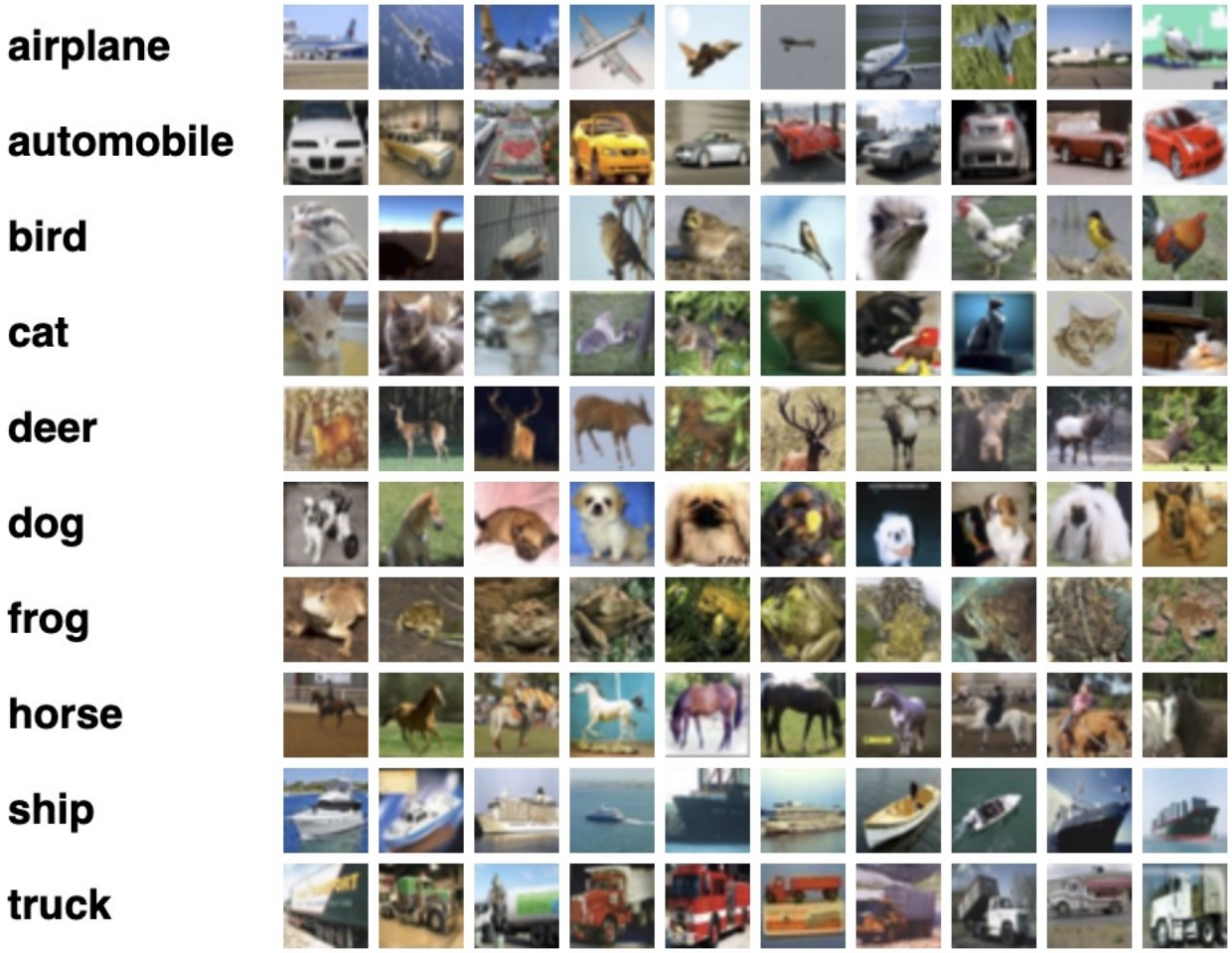
(continued from previous page)

```
        #   which each connect to the previous 512 neurons
        # - A log softmax layer which converts the 'unbounded' activations to the domain␣
↪[-inf, 0]
        #   (log softmax is numerically more stable than a regular softmax)
        #   (why we use the softmax in any case instead of simply normalizing the output␣
↪is a difficult question,
        #    see Bishop 2006)
        # Which translates to:

        self.layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(input_size, 512),
            nn.Linear(512, 10),
            nn.LogSoftmax(dim=1)
        )

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

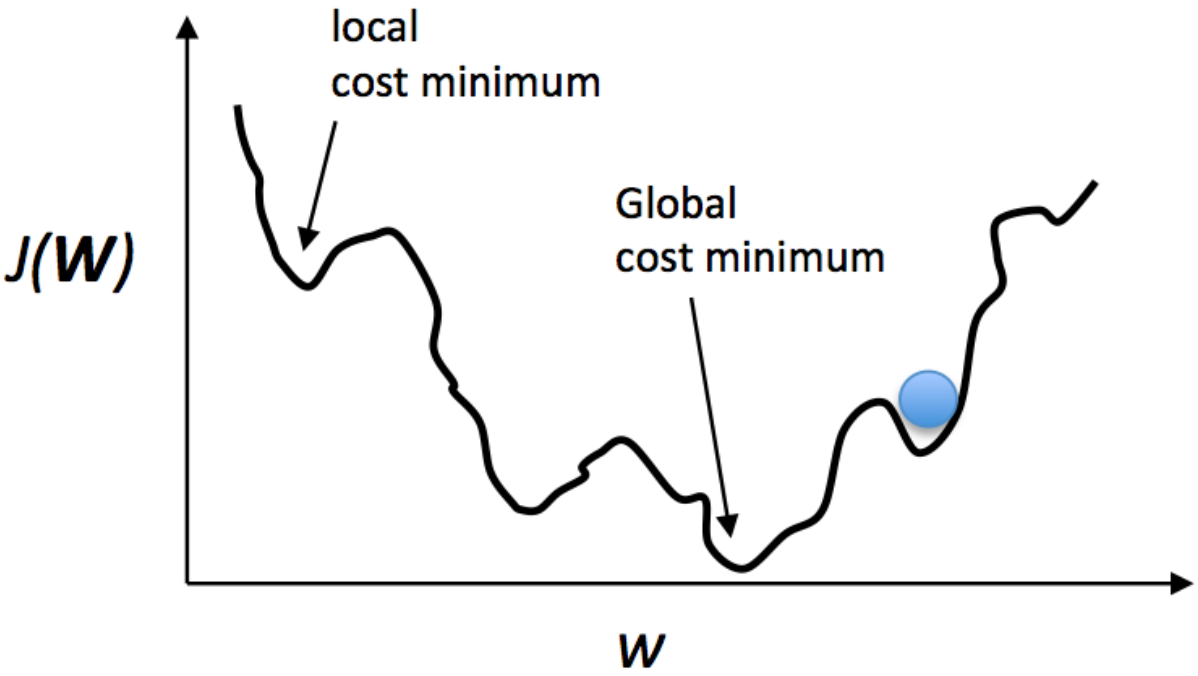
2.6.4 What we want to do to train this model; gradient descent:

Our predictions are probability distributions across the ten different digits (e.g. "we're 80% confident this image is a 3, 10% sure it's an 8, 5% it's a 2, etc."), and the target is a probability distribution with 100% for the correct category, and 0 for everything else. The cross-entropy is a measure of how different your predicted distribution is from the target distribution.

he optimizer helps determine how quickly the model learns through gradient descent. The rate at which descends a gradient is called the learning rate.



So are smaller learning rates better? Not quite! It's important for an optimizer not to get stuck in local minima while neglecting the global minimum of the loss function. Sometimes that means trying a larger learning rate to jump out of a local minimum.



Although this image is not entirely correct: it doesn't exist in reality. Loss manifolds are complicated... but let's not think about that too hard right now, instead:

2.6.5 What we need to train this model under the hood...

- The calculation and retention of computational graphs when you call model.forward()
- The calculation of gradients when we calculate the loss and call loss.backwards()
- The updating of model parameters when we call optimizer.step()

2.6.6 Take a look at the train loop below!

```
[ ]: def train(model, device, train_loader, optimizer, epoch, log_interval=10):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        # move data and target to the gpu, if available and used
        data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
↪target))

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()

        accuracy = metrics.accuracy(output, target)
```

(continues on next page)

(continued from previous page)

```
        if batch_idx % log_interval == 0:
            print(
                f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                f'\tLoss: {loss.detach().item():.6f}'
                f'\tAccuracy: {accuracy.detach().item():.2f}'
            )

        yield loss.detach().item(), accuracy.detach().item()
```

2.6.7 We provide the test loop

[ ]:
```
@torch.no_grad()
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0

    for data, target in test_loader:
        # move data and target to the gpu, if available and used
        data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
↪target))

        # get model output
        output = model(data)

        # calculate loss
        test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch␣
↪loss

        # get most likely class label
        pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-
↪probability

        # count the number of correct predictions
        correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100 * correct / len(test_loader.dataset)))

    yield test_loss, correct / len(test_loader.dataset)
```

2.6.8 PyTorch boilerplate

[ ]:
```
use_cuda = torch.cuda.is_available()
print(f"CUDA is {'' if use_cuda else 'not '}available")
device = torch.device("cuda" if use_cuda else "cpu")

if use_cuda:
    torch.cuda.set_per_process_memory_fraction(0.22)

cpu_count = len(os.sched_getaffinity(0))
```

2.6.9 Function to create train/validation curve plots

[ ]:
```
def plot_metric_curve(
    train_metric: Sequence[float],
    val_metric: Sequence[float],
    n_epochs: int,
    metric_name: str, # Label of the y-axis, e.g. 'Accuracy'
    x_axis_name: str = 'Epoch'
):
    # create values for the x-axis
    train_steps, val_steps = map(
        lambda metric_values: np.linspace(start=0, stop=n_epochs, num=len(metric_
↪values)),
        (train_metric, val_metric)
    )

    plt.plot(train_steps, train_metric, label='train')
    plt.plot(val_steps, val_metric, label='validation')
    plt.title(f"{metric_name} vs. {x_axis_name}")
    plt.legend()
    plt.show()
```

2.6.10 Function to run training and testing loops

[ ]:
```
def fit(model, optimizer, n_epochs, device, train_loader, test_loader, log_interval):

    # get the validation loss and accuracy of the untrained model
    start_val_loss, start_val_acc = tuple(test(model, device, test_loader))[0]

    # don't mind the following train/test loop logic too much, if you want to know what's␣
↪happening, let us know :)
    # normally you would pass a logger to your train/test loops and log the respective␣
↪metrics there
    (train_loss, train_acc), (val_loss, val_acc) = map(lambda arr: np.asarray(arr).
↪transpose(2,0,1), zip(*[
        (
            [*train(model, device, train_loader, optimizer, epoch, log_interval)],
            [*test(model, device, test_loader)]
        )
```

(continues on next page)

(continued from previous page)

```
        for epoch in range(n_epochs)
    ]))

    # flatten the arrays
    train_loss, train_acc, val_loss, val_acc = map(np.ravel, (train_loss, train_acc, val_
↪loss, val_acc))

    # prepend the validation loss and accuracy of the untrained model
    val_loss, val_acc = (start_val_loss, *val_loss), (start_val_acc, *val_acc)

    plot_metric_curve(train_loss, val_loss, n_epochs, 'Loss')
    plot_metric_curve(train_acc, val_acc, n_epochs, 'Accuracy')
```

2.6.11 Training the model

This is the fun part!

The batch size determines over how much data per step is used to compute the loss function, gradients, and back propagation. Large batch sizes allow the network to complete it's training faster; however, there are other factors beyond training speed to consider.

Too large of a batch size reduces the variance of the update step, which may not always be useful for succesfully training the model due to underfitting.

Too small of a batch size increases the variance of the update step, which may lead the optimizer to miss the global minimum.

So a good batch size may take some trial and error to find (or hyperparameter optimization...)!

[ ]:
```
BATCH_SIZE = ...
LEARNING_RATE = ...
EPOCHS = ...

LOGGING_INTERVAL = 100

model = LinearClassifier().to(device)
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE)


transform = transforms.Compose([
    transforms.ToTensor(), # Creates the PyTorch tensors from the PIL images, and␣
↪normalizes them to the [0, 1] interval
    transforms.Normalize((0.1307,), (0.3081,))  # Normalizes the data to 0 mean and 1␣
↪standard deviation
])

train_loader, test_loader = (
    torch.utils.data.DataLoader(
        datasets.MNIST(DATA_PATH, train=train, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=train,
        num_workers=cpu_count
```

(continues on next page)

(continued from previous page)

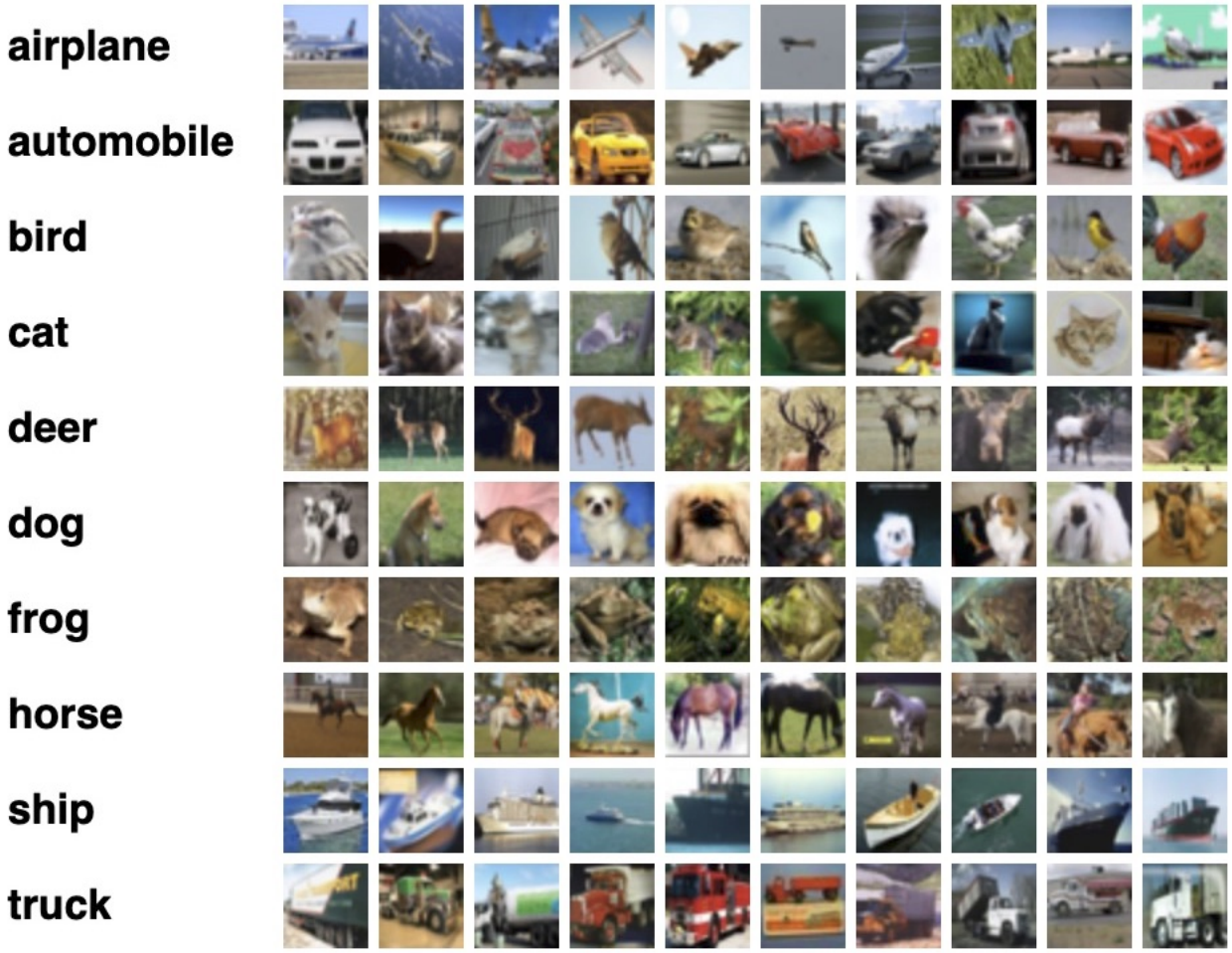
```
    )
    for train in (True, False)
)

fit(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_INTERVAL)
```

2.6.12 (Hopefully) your loss goes down, and your accuracy goes up!

Don't be dissuaded if your model doesn't train with your first set of hyperparameters, (efficient) hyperparameter optimization is still an unsolved research problem.

2.7 Making the classifier a Neural Network

We will build the following model:



To represent a model in PyTorch, you usually extend PyTorch's nn.Module class. There, we define our layers, and in its instance method *forward*, we define what happens when our model receives some input.

2.8 Define your model

[ ]:
```
class FCNN(nn.Module):
    def __init__(self, input_size=28*28):
        super().__init__()  # Python magic which initialises all relevant PyTorch␣
↪properties

        # Here, define your model!

        # We are going to need:
        # - A layer that flattens the input
        # - A fully connected layer which has 512 neurons who each connect to the full␣
↪28x28 input
        # - A non linear activation layer
        # - A fully connected layer which has 10 output neurons, each of which represent␣
↪an output class,
        #   which each connect to the previous 512 neurons
        # - A log softmax layer which converts the 'unbounded' activations to the domain␣
↪[-inf, 0]

        self.layers = nn.Sequential(
            ...
        )



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

2.9 Define your hyperparameters

[ ]:
```
BATCH_SIZE = ...
LEARNING_RATE = ...

EPOCHS = 10
LOGGING_INTERVAL = 100

model = FCNN().to(device)
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE)

transform = transforms.Compose([
    transforms.ToTensor(), # Creates the PyTorch tensors from the PIL images, and␣
↪normalizes them to the [0, 1] interval
    transforms.Normalize((0.1307,), (0.3081,))  # Normalizes the data to 0 mean and 1␣
↪standard deviation
])

train_loader, test_loader = (
    torch.utils.data.DataLoader(
```

(continues on next page)

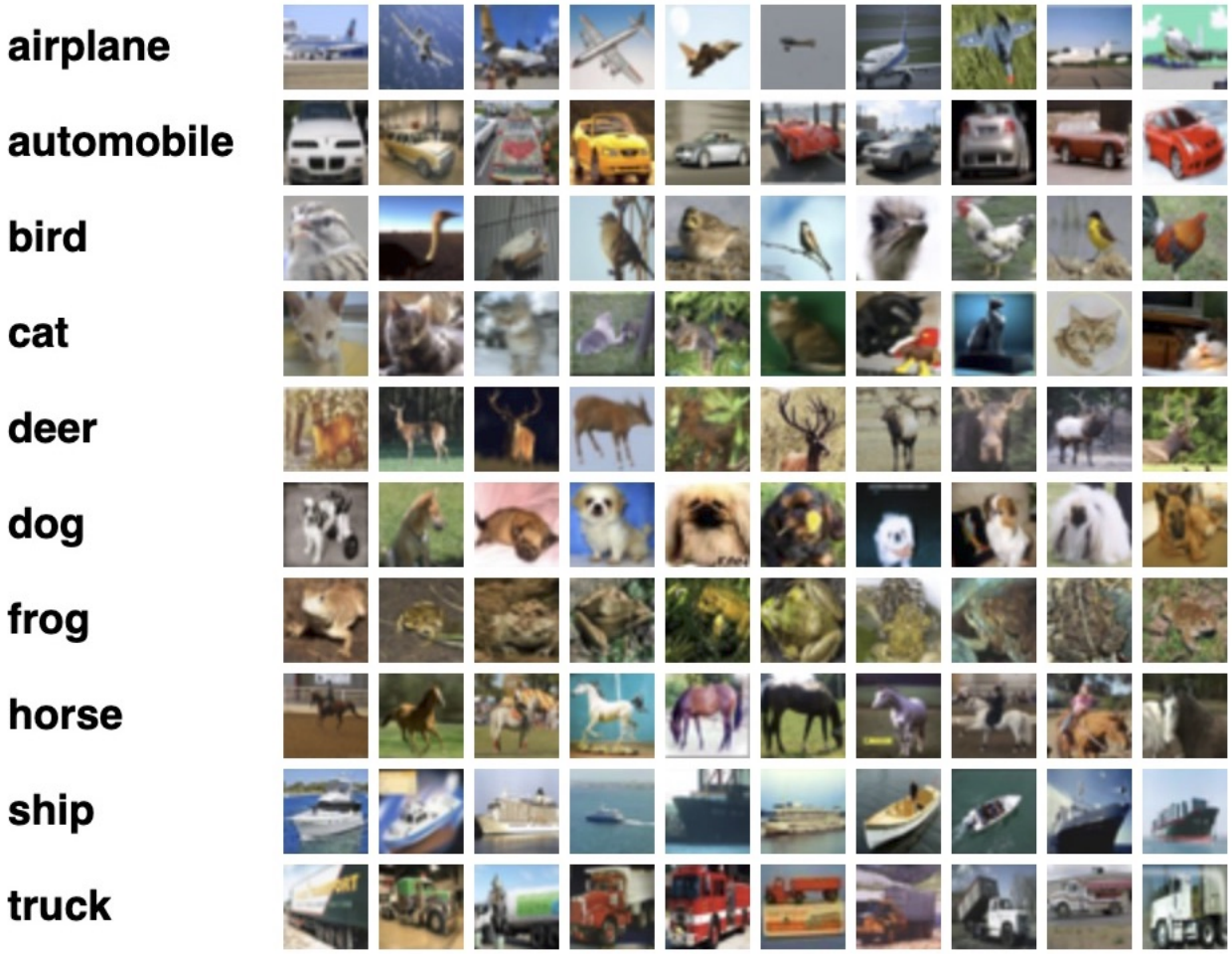
(continued from previous page)

```
        datasets.MNIST(DATA_PATH, train=train, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=train,
        num_workers=cpu_count
    )
    for train in (True, False)
)

fit(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_INTERVAL)
```

2.10 What about harder datasets?

MNIST is the easiest widely-used dataset as a computer vision toy-problem. One step above MNIST is CIFAR10, which contains images from 10 classes of objects.

The images are of size 32*32, and are not grayscale like MNIST, but contain 3 channels, one for each color in RGB: Red-Green-Blue.



[ ]:
```
BATCH_SIZE = ...
LEARNING_RATE = ...
EPOCHS = ...

LOGGING_INTERVAL = 100

# What should be the input size?
model = FCNN(input_size=...).to(device)
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalize the data to 0 mean and 1 standard deviation, now for all channels of RGB
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

train_loader, test_loader = (
    torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=train, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=True,
        shuffle=train
    )
    for train in (True, False)
)

fit(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_INTERVAL)
```

2.11 Ai, probably not the 90%+ accuracy we saw with MNIST!

What can you do about this? Try something out! For example: - Bigger model - Different optimizer - Different learning rate - More epochs - Data augmentation

Each methods has its upsides and downsides, think about what these are before changing something!

But maybe we can change the model to incorporate information about the data before even starting learning… ### Next chapter: CNNs; improving Fully-Connected networks with prior knowledge

[ ]:
```
import os
from typing import Sequence, Tuple

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchmetrics.functional as metrics
import numpy as np
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

%matplotlib inline
```

(continues on next page)

(continued from previous page)

```
DATA_PATH = os.getenv('TEACHER_DIR', os.getcwd()) + '/JHL_data'
```

2.12 Introducing Convolution! What is it?

Before, we built a network that accepts the normalized pixel values of each value and operates soley on those values. What if we could instead feed different features (e.g. curvature, edges) of each image into a network, and have the network learn which features are important for classifying an image?

This possible through convolution! Convolution applies kernels (filters) that traverse through each image and generate feature maps, see:

https://miro.medium.com/max/500/1*GcI7G-JLAQiEoCON7xFbhg.gif

In the above example, the image is a 5 x 5 matrix and the kernel going over it is a 3 x 3 matrix. A dot product operation takes place between the image and the kernel and the convolved feature is generated. Each kernel in a CNN learns a different characteristic of an image.

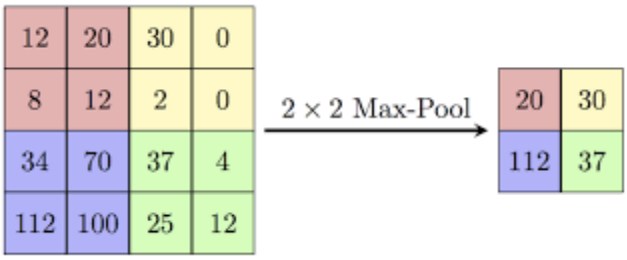
2.12.1 In the olden days:

Kernels are often used in photoediting software to apply blurring, edge detection, sharpening, etc.

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

2.12.2 Now:

Kernels in deep learning networks are used in similar ways, i.e. highlighting some feature. Combined with a system called max pooling, the non-highlighted elements are discarded from each feature map, leaving only the features of interest, reducing the number of learned parameters, and decreasing the computational cost (e.g. system memory).



We can also take convolutions of convolutions – we can stack as many convolutions as we want, as long as there are enough pixels to fit a kernel.

Warning: What you may find down there in those deep convolutions may not appear recognizable to you, see also: https://distill.pub/2019/activation-atlas/

2.13 How to continue?

We will try to classify CIFAR10 with our own CNN, re-using our train/test loops

[ ]:
```
def train(model, device, train_loader, optimizer, epoch, log_interval=10):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        # move data and target to the gpu, if available and used
        data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
↪target))

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
```

(continues on next page)

(continued from previous page)

```
        optimizer.step()

        accuracy = metrics.accuracy(output, target)

        if batch_idx % log_interval == 0:
            print(
                f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                f'\tLoss: {loss.detach().item():.6f}'
                f'\tAccuracy: {accuracy.detach().item():.2f}'
            )

        yield loss.detach().item(), accuracy.detach().item()


@torch.no_grad()
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0

    for data, target in test_loader:
        # move data and target to the gpu, if available and used
        data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
↪target))

        # get model output
        output = model(data)

        # calculate loss
        test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch␣
↪loss

        # get most likely class label
        pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-
↪probability

        # count the number of correct predictions
        correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100 * correct / len(test_loader.dataset)))

    yield test_loss, correct / len(test_loader.dataset)
```

[3]:
```
def plot_metric_curve(
    train_metric: Sequence[float],
    val_metric: Sequence[float],
    n_epochs: int,
```

(continues on next page)

(continued from previous page)

```
    metric_name: str, # Label of the y-axis, e.g. 'Accuracy'
    x_axis_name: str = 'Epoch'
):
    # create values for the x-axis
    train_steps, val_steps = map(
        lambda metric_values: np.linspace(start=0, stop=n_epochs, num=len(metric_
↪values)),
        (train_metric, val_metric)
    )

    plt.plot(train_steps, train_metric, label='train')
    plt.plot(val_steps, val_metric, label='validation')
    plt.title(f"{metric_name} vs. {x_axis_name}")
    plt.legend()
    plt.show()
```

[4]:
```
def fit(model, optimizer, n_epochs, device, train_loader, test_loader, log_interval):

    # get the validation loss and accuracy of the untrained model
    start_val_loss, start_val_acc = tuple(test(model, device, test_loader))[0]

    # don't mind the following train/test loop logic too much, if you want to know what's␣
↪happening, let us know :)
    # normally you would pass a logger to your train/test loops and log the respective␣
↪metrics there
    (train_loss, train_acc), (val_loss, val_acc) = map(lambda arr: np.asarray(arr).
↪transpose(2,0,1), zip(*[
        (
            [*train(model, device, train_loader, optimizer, epoch, log_interval)],
            [*test(model, device, test_loader)]
        )
        for epoch in range(n_epochs)
    ]))

    # flatten the arrays
    train_loss, train_acc, val_loss, val_acc = map(np.ravel, (train_loss, train_acc, val_
↪loss, val_acc))

    # prepend the validation loss and accuracy of the untrained model
    val_loss, val_acc = (start_val_loss, *val_loss), (start_val_acc, *val_acc)

    plot_metric_curve(train_loss, val_loss, n_epochs, 'Loss')
    plot_metric_curve(train_acc, val_acc, n_epochs, 'Accuracy')
```

2.14 Defining the model

Moving from a FCNN to a CNN, we split our model in two: a feature extractor and a classifier.
The classifier will be no different to the FCNN: Some linear layers with a non-linearity in-between.
The feature extractor will be our convolutional layers, with a downsampling operation after each layer; in our case max-pooling.
(Which you shouldn't normally use unless you have a very good reason to! Take it from Saint Geoff: https://mirror2image.wordpress.com/2014/11/11/geoffrey-hinton-on-max-pooling-reddit-ama/))

[ ]:
```
class CIFAR10CNN(nn.Module):
    def __init__(self):
        super().__init__()

        # 4 convolution layers, with a non-linear activation after each.
        # maxpooling after the activations of the 2nd, 3rd, and 4th conv layers
        # 2 dense layers for classification
        # log_softmax
        #
        # As for the number of channels of each layers, try to experiment!

        self.feature_extractor = nn.Sequential(
            ...
        )

        # in_features of the first layer should be the product of the output shape of␣
↪your feature extractor!
        # E.g. if the output of your feature extractor has size (batch x 128 x 4 x 4),␣
↪in_features = 128*4*4=2048
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=..., out_features=...),
            nn.ReLU(),
            nn.Linear(in_features=..., out_features=10),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, x):
        features = self.feature_extractor(x)

        return self.classifier(features)
```

[6]:
```
use_cuda = torch.cuda.is_available()
print(f"CUDA is {'' if use_cuda else 'not '}available")
device = torch.device("cuda" if use_cuda else "cpu")

if use_cuda:
    torch.cuda.set_per_process_memory_fraction(0.22)

cpu_count = len(os.sched_getaffinity(0))
```

```
CUDA is available
```

[ ]:
```
# We can Re-use our train and test loops

BATCH_SIZE = ...
LEARNING_RATE = ...
EPOCHS = ...

LOGGING_INTERVAL = 10

model = CIFAR10CNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalize the data to 0 mean and 1 standard deviation, now for all channels of RGB
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

train_loader, test_loader = (
    torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=train, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=train,
        num_workers=cpu_count
    )
    for train in (True, False)
)

fit(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_INTERVAL)
```

Achieving ~75% validation accuracy with our simple CNN Classifier should be possible, how did you do? Let us know!

- Do you see a difference in train accuracy and validation accuracy?
- How does the difference between training and validation accuracy change over time?

The State-of-the-art is currently 99.5% accuracy! (See https://paperswithcode.com/sota/image-classification-on-cifar-10)

One of the areas of improvement is that our model is still not very deep, modern models usually range from 18-50 layers. However, after around ~15 layers, you start to run into some issues with information propagation through your model: the gradient of the loss is not able to reach the first few layers of the model.

Checkout https://arxiv.org/pdf/1512.03385.pdf for the seminal paper fixing this issue.

[3]:
```
import os
from typing import Sequence, Tuple

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
```

(continues on next page)

(continued from previous page)

```
import torch.optim as optim
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
import torchmetrics.functional as metrics

%matplotlib inline

DATA_PATH = os.getenv('TEACHER_DIR', os.getcwd()) + '/JHL_data'
```

2.15 Introduction to Deep Learning with PyTorch

To help you understand the fundamentals of deep learning, this demo will walk through the basic steps of building a toy model for classifying handwritten numbers with accuracies surpassing 95%. This model will be a basic fully-connected neural network.

2.15.1 The Task for the Neural Network

Our goal is to construct and train an artificial neural network on thousands of images of handwritten digits so that it may successfully identify others when presented. The data that will be incorporated is the MNIST database which contains 60,000 images for training and 10,000 test images.

Loading Training Data

The MNIST dataset is conveniently bundled within Torch Vision, and we can easily take a look at some of its features.

2.15.2 Lets take a look at our input data

[4]:
```
example_trainset = datasets.MNIST(root=DATA_PATH, train=True, transform=transforms.
↪ToTensor(), download=True)

example_image, example_label = next(iter(example_trainset))

print(f"Input data shape: {example_image.shape}")
example_image = example_image[0] # removing first dimension

print(f"Input label: {example_label}")

plt.imshow(example_image, cmap='gray')
plt.show()
```

```
2.1%
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to /project/
↪jhlsrf016/JHL_data/MNIST/raw/train-images-idx3-ubyte.gz
```

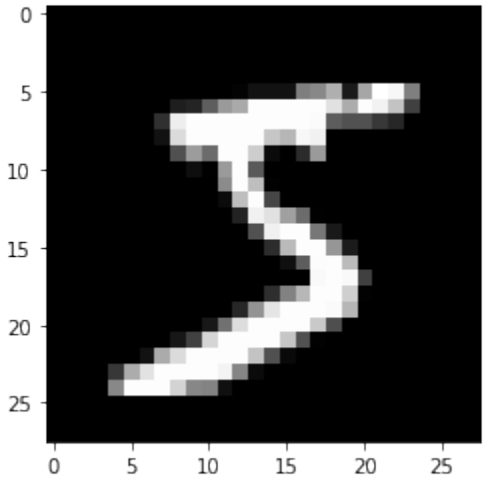
```
31.0%IOPub message rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_msg_rate_limit`.

Current values:
NotebookApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
NotebookApp.rate_limit_window=3.0 (secs)

85.7%IOPub message rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_msg_rate_limit`.

Current values:
NotebookApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
NotebookApp.rate_limit_window=3.0 (secs)

100.0%
112.7%
```

```
Extracting /project/jhlsrf016/JHL_data/MNIST/raw/t10k-images-idx3-ubyte.gz to /project/
↪jhlsrf016/JHL_data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to /project/
↪jhlsrf016/JHL_data/MNIST/raw/t10k-labels-idx1-ubyte.gz
```

(continues on next page)

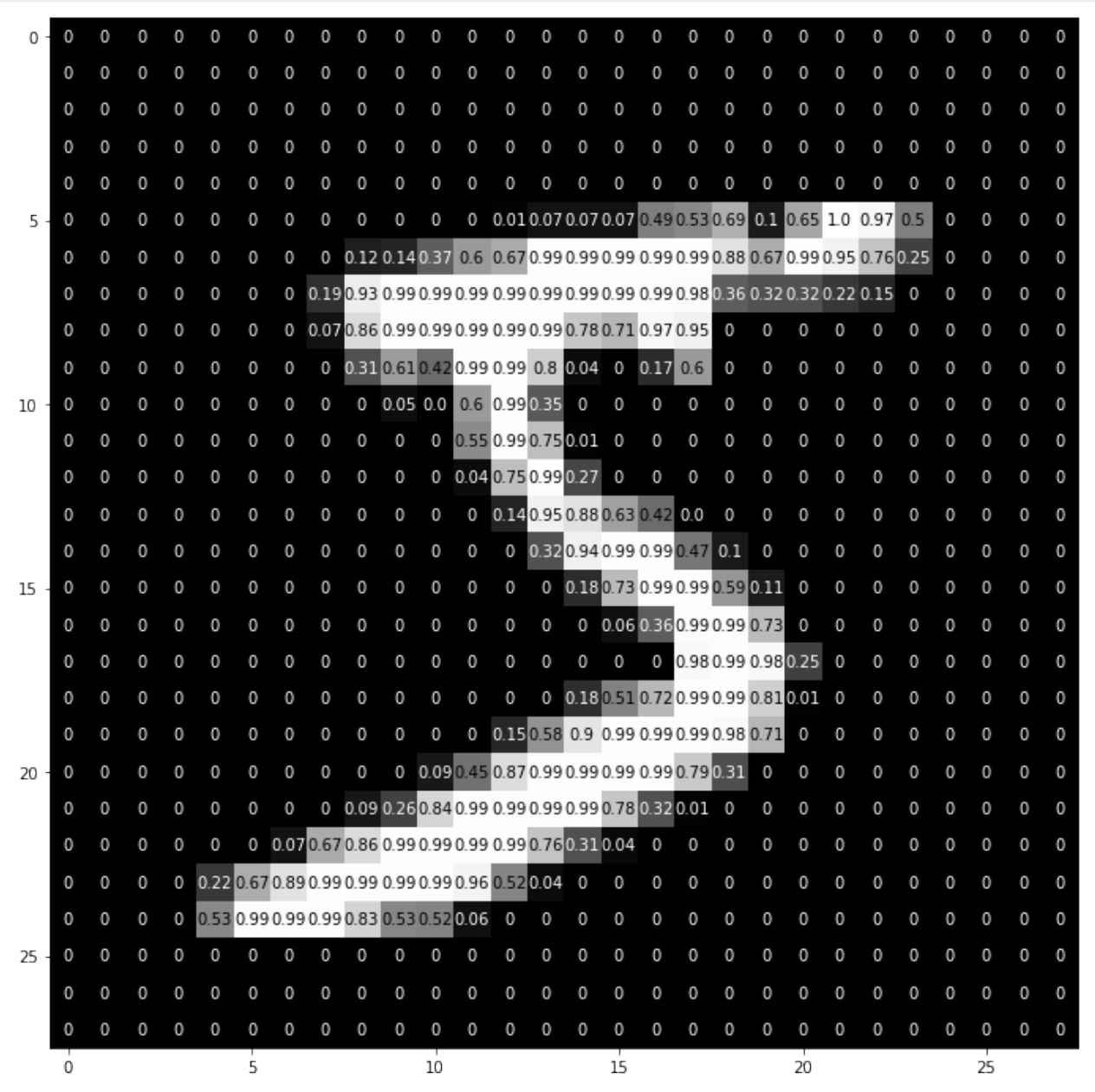
(continued from previous page)

```
Extracting /project/jhlsrf016/JHL_data/MNIST/raw/t10k-labels-idx1-ubyte.gz to /project/
↪jhlsrf016/JHL_data/MNIST/raw

Input data shape: torch.Size([1, 28, 28])
Input label: 5
```



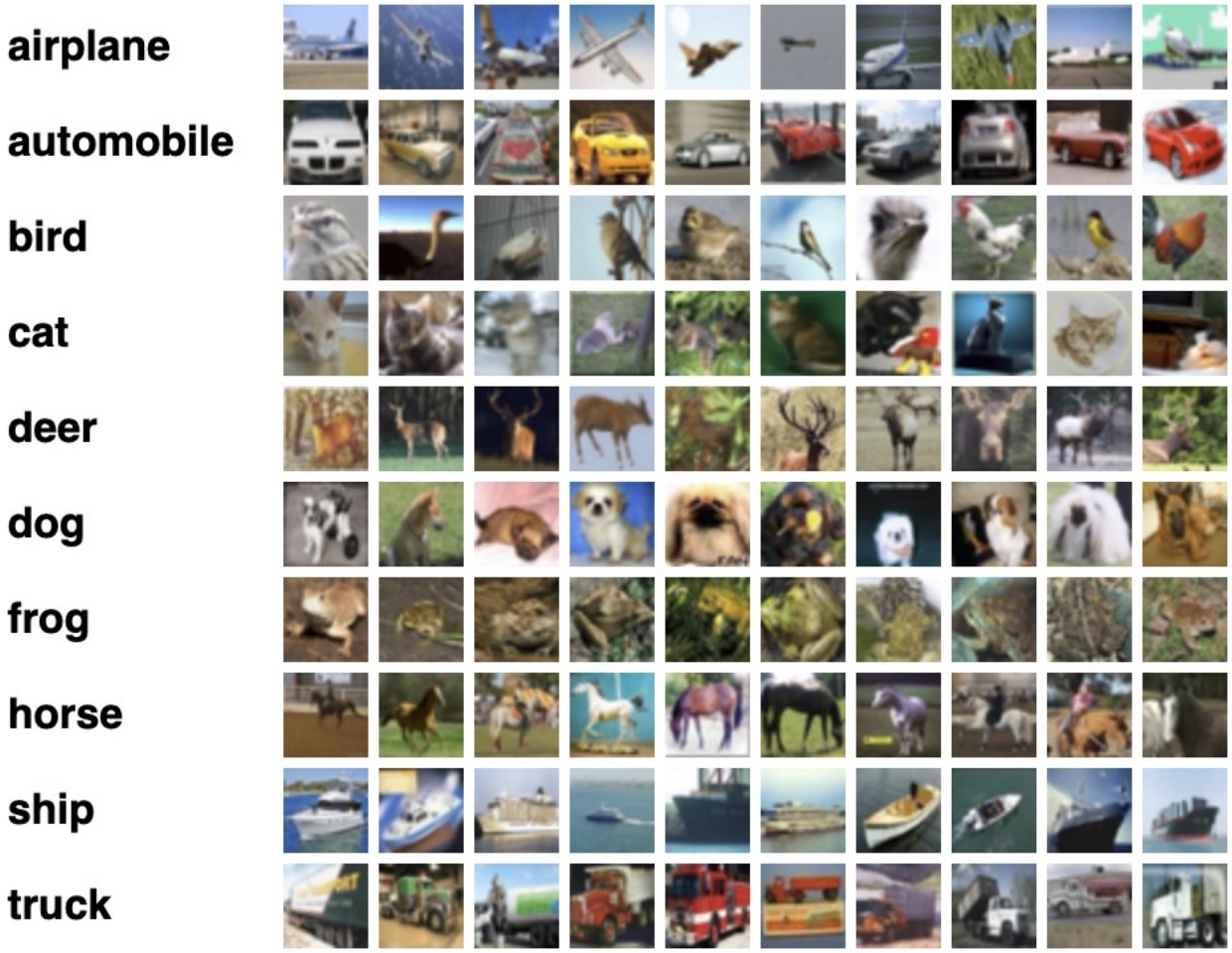
2.15.3 Let's plot the numerical representation of the image so that we can see what the model sees as input!

[5]:
```
fig = plt.figure(figsize = (12,12))
ax = fig.add_subplot(111)
ax.imshow(example_image, cmap='gray')
width, height = example_image.shape
thresh = example_image.max()/2.5
for x in range(width):
    for y in range(height):
        val = round(example_image[x][y].item(), 2) if example_image[x][y] != 0 else 0
        ax.annotate(str(val), xy=(y, x),
                    horizontalalignment='center',
                    verticalalignment='center',
                    color='white' if example_image[x][y]<thresh else 'black')
```



Formatting the input data layer Instead of a 28 x 28 matrix, we build our network to accept a 784-length vector.

Each image needs to be then reshaped (or flattened) into a vector.



This will be the first layer to the model we are going to build.

[6]:
```
class LinearClassifier(nn.Module):
    def __init__(self, input_size=28*28):
        super().__init__()  # Python magic which initialises all relevant PyTorch␣
↪properties

        # Defining the model:
        # - A layer that flattens the input
        # - A fully connected layer which has 512 neurons who each connect to the full␣
↪28x28 input
        # - A fully connected layer which has 10 neurons, each of which represent an␣
↪output class,
        #   which each connect to the previous 512 neurons
        # - A log softmax layer which converts the 'unbounded' activations to the domain␣
↪[-inf, 0]
        #   (log softmax is numerically more stable than a regular softmax)
        #   (why we use the softmax in any case instead of simply normalizing the output␣
↪is a difficult question,
        #    see Bishop 2006)
        # Which translates to:
```

(continues on next page)

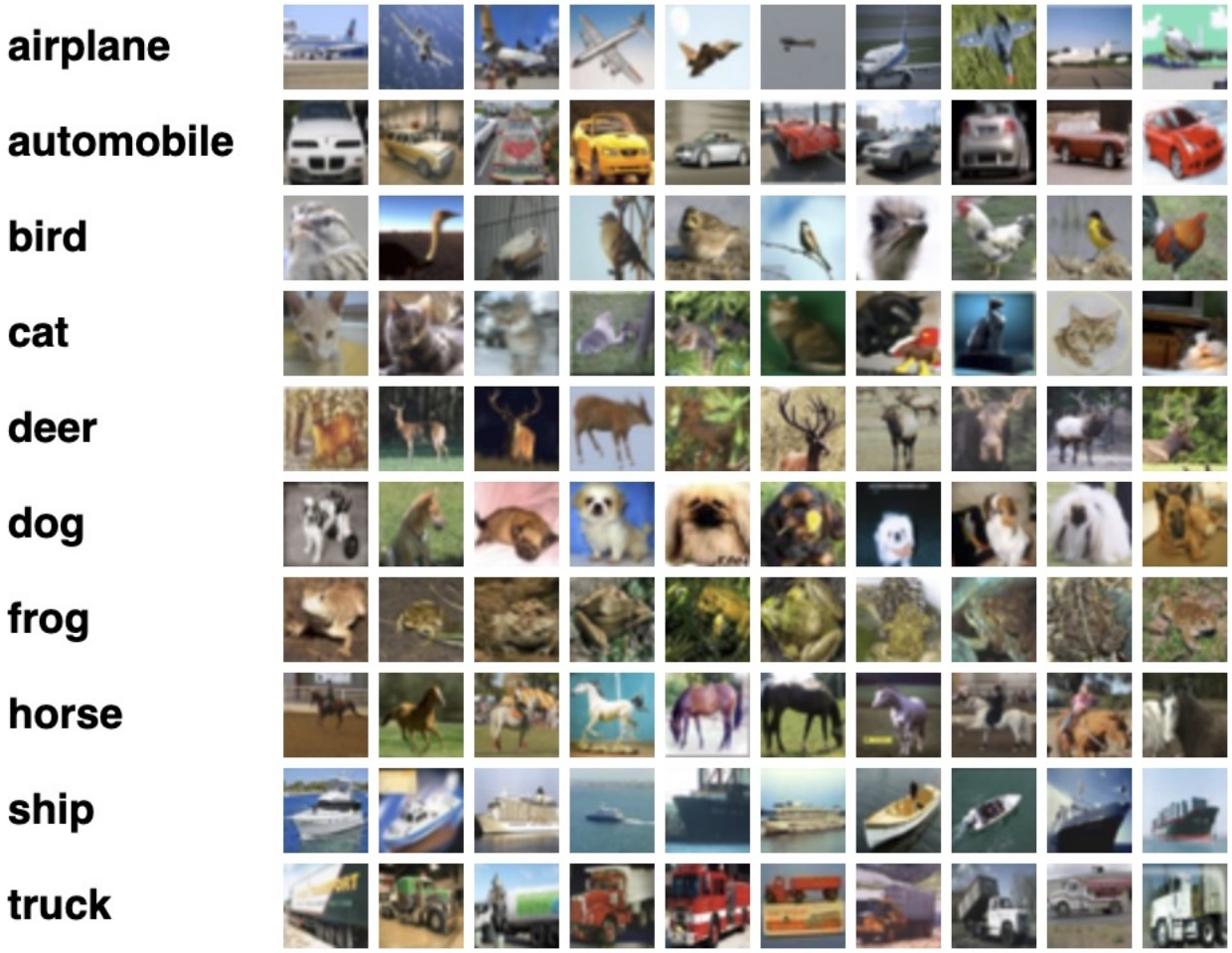
(continued from previous page)

```
        self.layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(input_size, 512),
            nn.Linear(512, 10),
            nn.LogSoftmax(dim=1)
        )

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

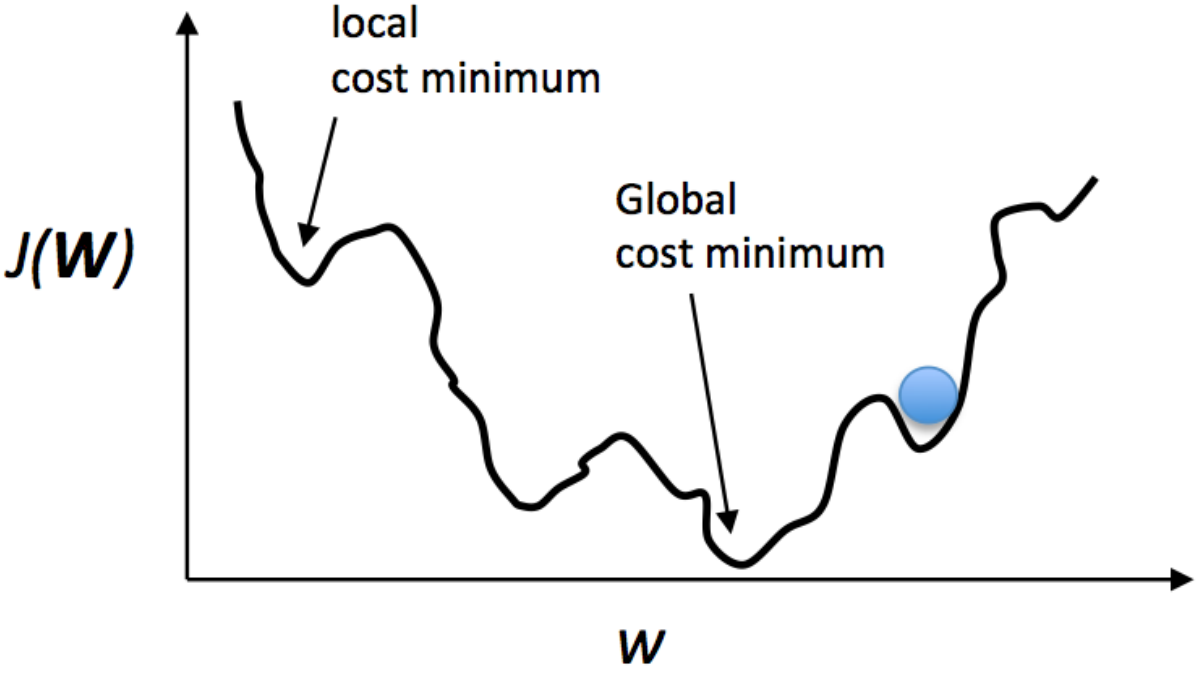
2.15.4 What we want to do to train this model; gradient descent:

Our predictions are probability distributions across the ten different digits (e.g. "we're 80% confident this image is a 3, 10% sure it's an 8, 5% it's a 2, etc."), and the target is a probability distribution with 100% for the correct category, and 0 for everything else. The cross-entropy is a measure of how different your predicted distribution is from the target distribution.

he optimizer helps determine how quickly the model learns through gradient descent. The rate at which descends a gradient is called the learning rate.



So are smaller learning rates better? Not quite! It's important for an optimizer not to get stuck in local minima while neglecting the global minimum of the loss function. Sometimes that means trying a larger learning rate to jump out of a local minimum.



Although this image is not entirely correct: it doesn't exist in reality. Loss manifolds are complicated...but let's not think about that too hard right now, instead:

2.15.5 What we need to train this model under the hood...

- The calculation and retention of computational graphs when you call model.forward()
- The calculation of gradients when we calculate the loss and call loss.backwards()
- The updating of model parameters when we call optimizer.step()

2.15.6 Take a look at the train loop below!

[7]:
```
def train(model, device, train_loader, optimizer, epoch, log_interval=10):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        # move data and target to the gpu, if available and used
        data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
↪target))

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

(continues on next page)

(continued from previous page)

```
        accuracy = metrics.accuracy(output, target)

        if batch_idx % log_interval == 0:
            print(
                f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                f'\tLoss: {loss.detach().item():.6f}'
                f'\tAccuracy: {accuracy.detach().item():.2f}'
            )

        yield loss.detach().item(), accuracy.detach().item()
```

2.15.7 We provide the test loop

[8]:
```
@torch.no_grad()
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0

    for data, target in test_loader:
        # move data and target to the gpu, if available and used
        data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
↪target))

        # get model output
        output = model(data)

        # calculate loss
        test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch␣
↪loss

        # get most likely class label
        pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-
↪probability

        # count the number of correct predictions
        correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100 * correct / len(test_loader.dataset)))

    yield test_loss, correct / len(test_loader.dataset)
```

2.15.8 PyTorch boilerplate

[9]:
```
use_cuda = torch.cuda.is_available()
print(f"CUDA is {'' if use_cuda else 'not '}available")
device = torch.device("cuda" if use_cuda else "cpu")

if use_cuda:
    torch.cuda.set_per_process_memory_fraction(0.22)

cpu_count = len(os.sched_getaffinity(0))
```

```
CUDA is available
```

2.15.9 Function to plot train/validation curves

[10]:
```
def plot_metric_curve(
    train_metric: Sequence[float],
    val_metric: Sequence[float],
    n_epochs: int,
    metric_name: str, # Label of the y-axis, e.g. 'Accuracy'
    x_axis_name: str = 'Epoch'
):
    # create values for the x-axis
    train_steps, val_steps = map(
        lambda metric_values: np.linspace(start=0, stop=n_epochs, num=len(metric_
↪values)),
        (train_metric, val_metric)
    )

    plt.plot(train_steps, train_metric, label='train')
    plt.plot(val_steps, val_metric, label='validation')
    plt.title(f"{metric_name} vs. {x_axis_name}")
    plt.legend()
    plt.show()
```

2.15.10 Function to run training and testing loops

[11]:
```
def fit(model, optimizer, n_epochs, device, train_loader, test_loader, log_interval):

    # get the validation loss and accuracy of the untrained model
    start_val_loss, start_val_acc = tuple(test(model, device, test_loader))[0]

    # don't mind the following train/test loop logic too much, if you want to know what's␣
↪happening, let us know :)
    # normally you would pass a logger to your train/test loops and log the respective␣
↪metrics there
    (train_loss, train_acc), (val_loss, val_acc) = map(lambda arr: np.asarray(arr).
↪transpose(2,0,1), zip(*[
        (
            [*train(model, device, train_loader, optimizer, epoch, log_interval)],
```

(continues on next page)

(continued from previous page)

```
            [*test(model, device, test_loader)]
        )
        for epoch in range(n_epochs)
    ]))

    # flatten the arrays
    train_loss, train_acc, val_loss, val_acc = map(np.ravel, (train_loss, train_acc, val_
↪loss, val_acc))

    # prepend the validation loss and accuracy of the untrained model
    val_loss, val_acc = (start_val_loss, *val_loss), (start_val_acc, *val_acc)

    plot_metric_curve(train_loss, val_loss, n_epochs, 'Loss')
    plot_metric_curve(train_acc, val_acc, n_epochs, 'Accuracy')
```

2.15.11 Training the model

This is the fun part!

The batch size determines over how much data per step is used to compute the loss function, gradients, and back propagation. Large batch sizes allow the network to complete it's training faster; however, there are other factors beyond training speed to consider.

Too large of a batch size reduces the variance of the update step, which may not always be useful for succesfully training the model due to underfitting.

Too small of a batch size increases the variance of the update step, which may lead the optimizer to miss the global minimum.

So a good batch size may take some trial and error to find (or hyperparameter optimization...)!

[12]:
```
BATCH_SIZE = 64
LEARNING_RATE = 0.001
EPOCHS = 10

LOGGING_INTERVAL = 100

model = LinearClassifier().to(device)
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE)

transform = transforms.Compose([
    transforms.ToTensor(), # Creates the PyTorch tensors from the PIL images, and␣
↪normalizes them to the [0, 1] interval
    transforms.Normalize((0.1307,), (0.3081,))  # Normalizes the data to 0 mean and 1␣
↪standard deviation
])

train_loader, test_loader = (
    torch.utils.data.DataLoader(
        datasets.MNIST(DATA_PATH, train=train, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=train,
```

(continues on next page)

(continued from previous page)

```
        num_workers=cpu_count
    )
    for train in (True, False)
)

fit(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_INTERVAL)
```

```
Test set: Average loss: 2.3064, Accuracy: 1174/10000 (12%)

Train Epoch: 0 [0/60000 (0%)]  Loss: 2.331443  Accuracy: 0.06
Train Epoch: 0 [6400/60000 (11%)]      Loss: 1.728720  Accuracy: 0.58
Train Epoch: 0 [12800/60000 (21%)]     Loss: 1.305417  Accuracy: 0.78
Train Epoch: 0 [19200/60000 (32%)]     Loss: 1.250137  Accuracy: 0.73
Train Epoch: 0 [25600/60000 (43%)]     Loss: 1.215200  Accuracy: 0.66
Train Epoch: 0 [32000/60000 (53%)]     Loss: 0.970949  Accuracy: 0.80
Train Epoch: 0 [38400/60000 (64%)]     Loss: 0.879034  Accuracy: 0.83
Train Epoch: 0 [44800/60000 (75%)]     Loss: 0.706373  Accuracy: 0.88
Train Epoch: 0 [51200/60000 (85%)]     Loss: 0.869866  Accuracy: 0.84
Train Epoch: 0 [57600/60000 (96%)]     Loss: 0.765989  Accuracy: 0.88

Test set: Average loss: 0.6818, Accuracy: 8479/10000 (85%)

Train Epoch: 1 [0/60000 (0%)]  Loss: 0.859519  Accuracy: 0.81
Train Epoch: 1 [6400/60000 (11%)]      Loss: 0.543142  Accuracy: 0.89
Train Epoch: 1 [12800/60000 (21%)]     Loss: 0.577082  Accuracy: 0.84
Train Epoch: 1 [19200/60000 (32%)]     Loss: 0.701044  Accuracy: 0.80
Train Epoch: 1 [25600/60000 (43%)]     Loss: 0.531838  Accuracy: 0.89
Train Epoch: 1 [32000/60000 (53%)]     Loss: 0.520050  Accuracy: 0.89
Train Epoch: 1 [38400/60000 (64%)]     Loss: 0.639744  Accuracy: 0.80
Train Epoch: 1 [44800/60000 (75%)]     Loss: 0.479699  Accuracy: 0.89
Train Epoch: 1 [51200/60000 (85%)]     Loss: 0.421309  Accuracy: 0.95
Train Epoch: 1 [57600/60000 (96%)]     Loss: 0.364844  Accuracy: 0.92

Test set: Average loss: 0.4993, Accuracy: 8795/10000 (88%)

Train Epoch: 2 [0/60000 (0%)]  Loss: 0.530136  Accuracy: 0.91
Train Epoch: 2 [6400/60000 (11%)]      Loss: 0.460276  Accuracy: 0.84
Train Epoch: 2 [12800/60000 (21%)]     Loss: 0.453302  Accuracy: 0.89
Train Epoch: 2 [19200/60000 (32%)]     Loss: 0.370793  Accuracy: 0.92
Train Epoch: 2 [25600/60000 (43%)]     Loss: 0.525292  Accuracy: 0.88
Train Epoch: 2 [32000/60000 (53%)]     Loss: 0.474298  Accuracy: 0.86
Train Epoch: 2 [38400/60000 (64%)]     Loss: 0.602194  Accuracy: 0.81
Train Epoch: 2 [44800/60000 (75%)]     Loss: 0.457901  Accuracy: 0.86
Train Epoch: 2 [51200/60000 (85%)]     Loss: 0.576913  Accuracy: 0.89
Train Epoch: 2 [57600/60000 (96%)]     Loss: 0.307910  Accuracy: 0.95

Test set: Average loss: 0.4312, Accuracy: 8892/10000 (89%)

Train Epoch: 3 [0/60000 (0%)]  Loss: 0.365533  Accuracy: 0.92
Train Epoch: 3 [6400/60000 (11%)]      Loss: 0.474275  Accuracy: 0.89
Train Epoch: 3 [12800/60000 (21%)]     Loss: 0.442904  Accuracy: 0.86
```

(continues on next page)

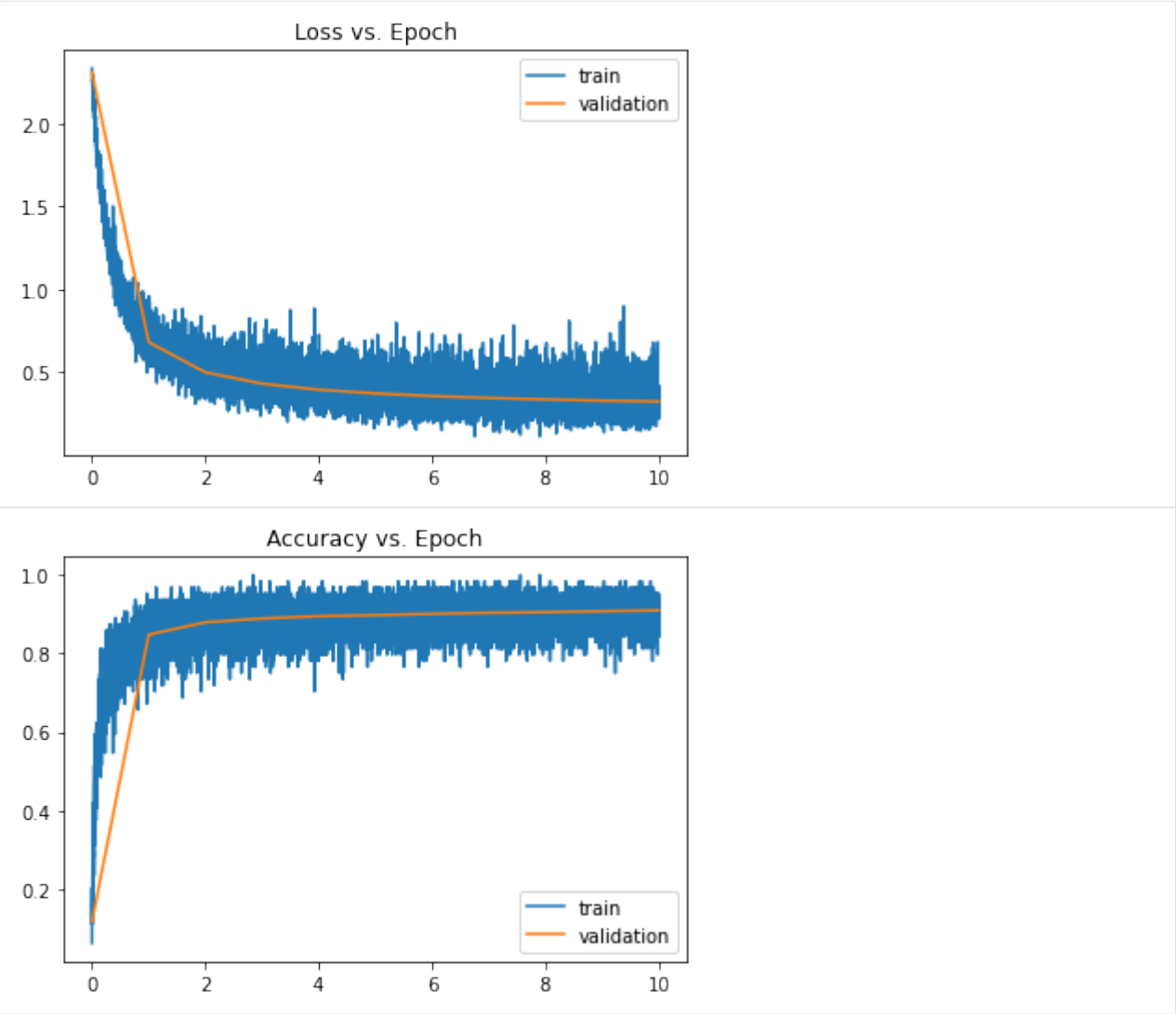
(continued from previous page)

```
Train Epoch: 3 [19200/60000 (32%)]     Loss: 0.469745  Accuracy: 0.84
Train Epoch: 3 [25600/60000 (43%)]     Loss: 0.350534  Accuracy: 0.91
Train Epoch: 3 [32000/60000 (53%)]     Loss: 0.417603  Accuracy: 0.88
Train Epoch: 3 [38400/60000 (64%)]     Loss: 0.421104  Accuracy: 0.83
Train Epoch: 3 [44800/60000 (75%)]     Loss: 0.394370  Accuracy: 0.86
Train Epoch: 3 [51200/60000 (85%)]     Loss: 0.542957  Accuracy: 0.89
Train Epoch: 3 [57600/60000 (96%)]     Loss: 0.422754  Accuracy: 0.88

Test set: Average loss: 0.3947, Accuracy: 8950/10000 (90%)

Train Epoch: 4 [0/60000 (0%)]  Loss: 0.537913  Accuracy: 0.80
Train Epoch: 4 [6400/60000 (11%)]      Loss: 0.498668  Accuracy: 0.92
Train Epoch: 4 [12800/60000 (21%)]     Loss: 0.298041  Accuracy: 0.94
Train Epoch: 4 [19200/60000 (32%)]     Loss: 0.472254  Accuracy: 0.83
Train Epoch: 4 [25600/60000 (43%)]     Loss: 0.678647  Accuracy: 0.73
Train Epoch: 4 [32000/60000 (53%)]     Loss: 0.429032  Accuracy: 0.89
Train Epoch: 4 [38400/60000 (64%)]     Loss: 0.360325  Accuracy: 0.91
Train Epoch: 4 [44800/60000 (75%)]     Loss: 0.429622  Accuracy: 0.86
Train Epoch: 4 [51200/60000 (85%)]     Loss: 0.426429  Accuracy: 0.88
Train Epoch: 4 [57600/60000 (96%)]     Loss: 0.423398  Accuracy: 0.88

Test set: Average loss: 0.3721, Accuracy: 8977/10000 (90%)

Train Epoch: 5 [0/60000 (0%)]  Loss: 0.489295  Accuracy: 0.83
Train Epoch: 5 [6400/60000 (11%)]      Loss: 0.299170  Accuracy: 0.92
Train Epoch: 5 [12800/60000 (21%)]     Loss: 0.375334  Accuracy: 0.91
Train Epoch: 5 [19200/60000 (32%)]     Loss: 0.354251  Accuracy: 0.92
Train Epoch: 5 [25600/60000 (43%)]     Loss: 0.565514  Accuracy: 0.89
Train Epoch: 5 [32000/60000 (53%)]     Loss: 0.336655  Accuracy: 0.91
Train Epoch: 5 [38400/60000 (64%)]     Loss: 0.319799  Accuracy: 0.91
Train Epoch: 5 [44800/60000 (75%)]     Loss: 0.357913  Accuracy: 0.91
Train Epoch: 5 [51200/60000 (85%)]     Loss: 0.239279  Accuracy: 0.95
Train Epoch: 5 [57600/60000 (96%)]     Loss: 0.545685  Accuracy: 0.83

Test set: Average loss: 0.3568, Accuracy: 9008/10000 (90%)

Train Epoch: 6 [0/60000 (0%)]  Loss: 0.408330  Accuracy: 0.88
Train Epoch: 6 [6400/60000 (11%)]      Loss: 0.386741  Accuracy: 0.89
Train Epoch: 6 [12800/60000 (21%)]     Loss: 0.249629  Accuracy: 0.94
Train Epoch: 6 [19200/60000 (32%)]     Loss: 0.365187  Accuracy: 0.86
Train Epoch: 6 [25600/60000 (43%)]     Loss: 0.359605  Accuracy: 0.91
Train Epoch: 6 [32000/60000 (53%)]     Loss: 0.263642  Accuracy: 0.92
Train Epoch: 6 [38400/60000 (64%)]     Loss: 0.359626  Accuracy: 0.84
Train Epoch: 6 [44800/60000 (75%)]     Loss: 0.394329  Accuracy: 0.88
Train Epoch: 6 [51200/60000 (85%)]     Loss: 0.297536  Accuracy: 0.94
Train Epoch: 6 [57600/60000 (96%)]     Loss: 0.338044  Accuracy: 0.89

Test set: Average loss: 0.3450, Accuracy: 9038/10000 (90%)

Train Epoch: 7 [0/60000 (0%)]  Loss: 0.258215  Accuracy: 0.91
Train Epoch: 7 [6400/60000 (11%)]      Loss: 0.144120  Accuracy: 0.98
Train Epoch: 7 [12800/60000 (21%)]     Loss: 0.360552  Accuracy: 0.89
```

(continues on next page)

(continued from previous page)

```
Train Epoch: 7 [19200/60000 (32%)]     Loss: 0.259858  Accuracy: 0.89
Train Epoch: 7 [25600/60000 (43%)]     Loss: 0.140562  Accuracy: 0.98
Train Epoch: 7 [32000/60000 (53%)]     Loss: 0.382180  Accuracy: 0.86
Train Epoch: 7 [38400/60000 (64%)]     Loss: 0.361994  Accuracy: 0.88
Train Epoch: 7 [44800/60000 (75%)]     Loss: 0.212859  Accuracy: 0.94
Train Epoch: 7 [51200/60000 (85%)]     Loss: 0.334832  Accuracy: 0.91
Train Epoch: 7 [57600/60000 (96%)]     Loss: 0.313229  Accuracy: 0.86

Test set: Average loss: 0.3365, Accuracy: 9053/10000 (91%)

Train Epoch: 8 [0/60000 (0%)]  Loss: 0.274177  Accuracy: 0.92
Train Epoch: 8 [6400/60000 (11%)]      Loss: 0.542565  Accuracy: 0.88
Train Epoch: 8 [12800/60000 (21%)]     Loss: 0.314171  Accuracy: 0.92
Train Epoch: 8 [19200/60000 (32%)]     Loss: 0.481625  Accuracy: 0.88
Train Epoch: 8 [25600/60000 (43%)]     Loss: 0.334137  Accuracy: 0.91
Train Epoch: 8 [32000/60000 (53%)]     Loss: 0.340529  Accuracy: 0.94
Train Epoch: 8 [38400/60000 (64%)]     Loss: 0.296860  Accuracy: 0.92
Train Epoch: 8 [44800/60000 (75%)]     Loss: 0.409278  Accuracy: 0.89
Train Epoch: 8 [51200/60000 (85%)]     Loss: 0.416559  Accuracy: 0.92
Train Epoch: 8 [57600/60000 (96%)]     Loss: 0.347065  Accuracy: 0.91

Test set: Average loss: 0.3289, Accuracy: 9078/10000 (91%)

Train Epoch: 9 [0/60000 (0%)]  Loss: 0.245768  Accuracy: 0.94
Train Epoch: 9 [6400/60000 (11%)]      Loss: 0.309143  Accuracy: 0.89
Train Epoch: 9 [12800/60000 (21%)]     Loss: 0.359546  Accuracy: 0.91
Train Epoch: 9 [19200/60000 (32%)]     Loss: 0.352461  Accuracy: 0.86
Train Epoch: 9 [25600/60000 (43%)]     Loss: 0.182498  Accuracy: 0.97
Train Epoch: 9 [32000/60000 (53%)]     Loss: 0.563132  Accuracy: 0.88
Train Epoch: 9 [38400/60000 (64%)]     Loss: 0.426583  Accuracy: 0.86
Train Epoch: 9 [44800/60000 (75%)]     Loss: 0.475921  Accuracy: 0.84
Train Epoch: 9 [51200/60000 (85%)]     Loss: 0.328519  Accuracy: 0.92
Train Epoch: 9 [57600/60000 (96%)]     Loss: 0.409453  Accuracy: 0.89

Test set: Average loss: 0.3234, Accuracy: 9099/10000 (91%)
```

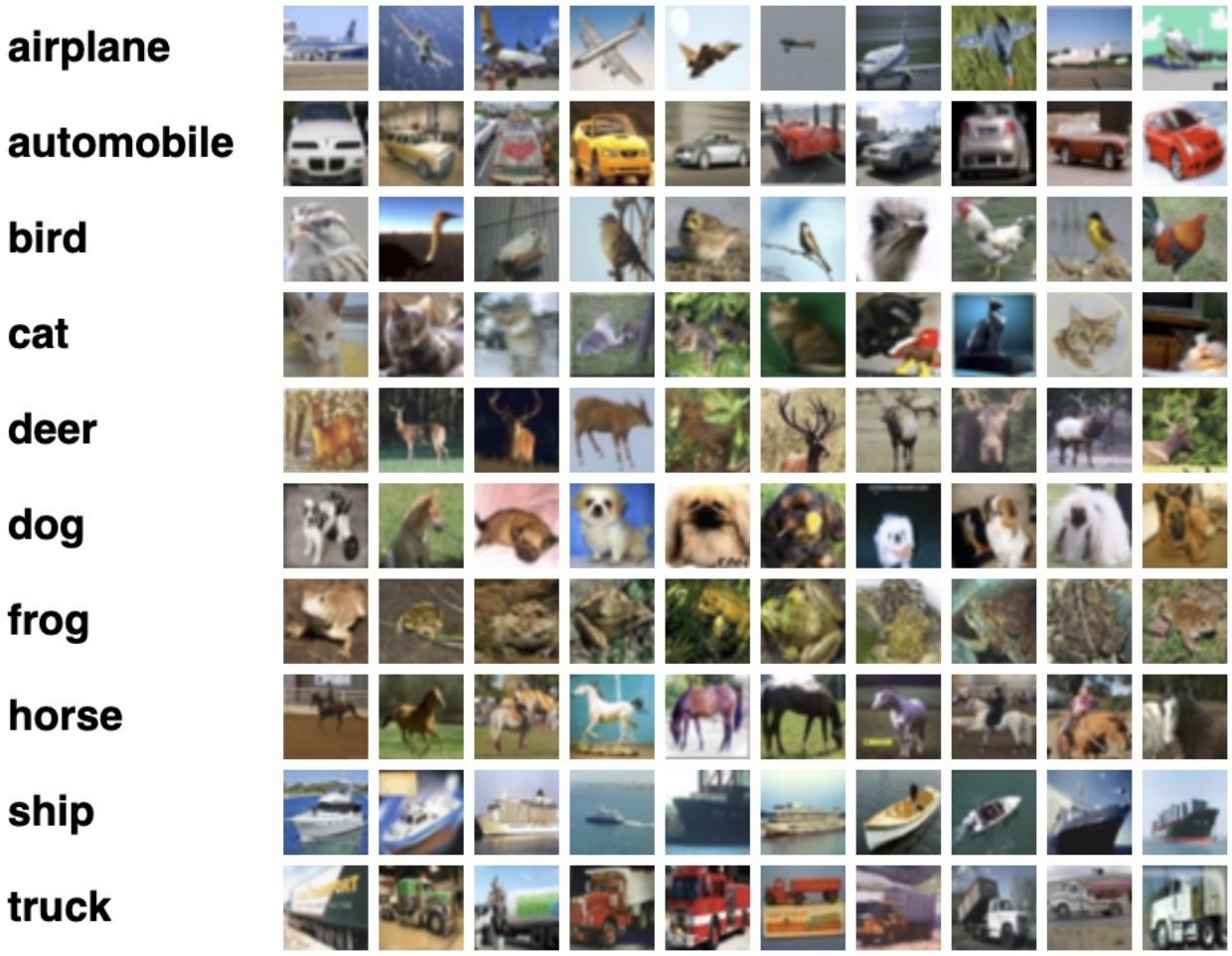


2.15.12 (Hopefully) your loss goes down, and your accuracy goes up!

Don't be dissuaded if your model doesn't train with your first set of hyperparameters, (efficient) hyperparameter optimization is still an unsolved research problem.

2.16 Making the classifier a Neural Network

We will build the following model:



To represent a model in PyTorch, you usually extend PyTorch's nn.Module class. There, we define our layers, and in its instance method *forward*, we define what happens when our model receives some input.

[13]:
```
class FCNN(nn.Module):
    def __init__(self, input_size=28*28):
        super().__init__()  # Python magic which initialises all relevant PyTorch␣
↪properties

        # Here, define your model!
        # We are going to need:

        self.layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(input_size, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.LogSoftmax(dim=1)
```

(continues on next page)

(continued from previous page)

```
        )
        # - A layer that flattens the input
        # - A fully connected layer which has 512 neurons who each connect to the full␣
↪28x28 input
        # - A non linear activation layer
        # - A fully connected layer which has 10 output neurons, each of which represent␣
↪an output class,
        #   which each connect to the previous 512 neurons
        # - A log softmax layer which converts the 'unbounded' activations to the domain␣
↪[-inf, 0]


    def forward(self, x):
        # data.shape = [batch_size, n_channels, height, width]
        # data.shape = [batch_size, n_channels, height*width]

        return self.layers(x)
```

2.17 Define your hyperparameters

[14]:
```
BATCH_SIZE = 64
EPOCHS = 10
LEARNING_RATE = 0.001
LOG_INTERVAL = 100

model = FCNN().to(device)
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE)

transform = transforms.Compose([
    transforms.ToTensor(), # Creates the PyTorch tensors from the PIL images, and␣
↪normalizes them to the [0, 1] interval
    transforms.Normalize((0.1307,), (0.3081,))  # Normalizes the data to 0 mean and 1␣
↪standard deviation
])

train_loader, test_loader = (
    torch.utils.data.DataLoader(
        datasets.MNIST(DATA_PATH, train=train, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=train,
        num_workers=cpu_count
    )
    for train in (True, False)
)

fit(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_INTERVAL)
```

(continues on next page)

(continued from previous page)

```
Test set: Average loss: 2.3291, Accuracy: 794/10000 (8%)

Train Epoch: 0 [0/60000 (0%)]  Loss: 2.345795  Accuracy: 0.08
Train Epoch: 0 [6400/60000 (11%)]      Loss: 2.082210  Accuracy: 0.39
Train Epoch: 0 [12800/60000 (21%)]     Loss: 1.937725  Accuracy: 0.53
Train Epoch: 0 [19200/60000 (32%)]     Loss: 1.669410  Accuracy: 0.69
Train Epoch: 0 [25600/60000 (43%)]     Loss: 1.547767  Accuracy: 0.69
Train Epoch: 0 [32000/60000 (53%)]     Loss: 1.315444  Accuracy: 0.73
Train Epoch: 0 [38400/60000 (64%)]     Loss: 1.201575  Accuracy: 0.77
Train Epoch: 0 [44800/60000 (75%)]     Loss: 1.090327  Accuracy: 0.81
Train Epoch: 0 [51200/60000 (85%)]     Loss: 1.150472  Accuracy: 0.83
Train Epoch: 0 [57600/60000 (96%)]     Loss: 0.972355  Accuracy: 0.84

Test set: Average loss: 0.9220, Accuracy: 8195/10000 (82%)

Train Epoch: 1 [0/60000 (0%)]  Loss: 0.902586  Accuracy: 0.78
Train Epoch: 1 [6400/60000 (11%)]      Loss: 0.781494  Accuracy: 0.94
Train Epoch: 1 [12800/60000 (21%)]     Loss: 0.741285  Accuracy: 0.88
Train Epoch: 1 [19200/60000 (32%)]     Loss: 0.754406  Accuracy: 0.86
Train Epoch: 1 [25600/60000 (43%)]     Loss: 0.631536  Accuracy: 0.89
Train Epoch: 1 [32000/60000 (53%)]     Loss: 0.783239  Accuracy: 0.80
Train Epoch: 1 [38400/60000 (64%)]     Loss: 0.598814  Accuracy: 0.88
Train Epoch: 1 [44800/60000 (75%)]     Loss: 0.714117  Accuracy: 0.86
Train Epoch: 1 [51200/60000 (85%)]     Loss: 0.646160  Accuracy: 0.84
Train Epoch: 1 [57600/60000 (96%)]     Loss: 0.601652  Accuracy: 0.91

Test set: Average loss: 0.5834, Accuracy: 8701/10000 (87%)

Train Epoch: 2 [0/60000 (0%)]  Loss: 0.594234  Accuracy: 0.81
Train Epoch: 2 [6400/60000 (11%)]      Loss: 0.517734  Accuracy: 0.89
Train Epoch: 2 [12800/60000 (21%)]     Loss: 0.628767  Accuracy: 0.84
Train Epoch: 2 [19200/60000 (32%)]     Loss: 0.600030  Accuracy: 0.88
Train Epoch: 2 [25600/60000 (43%)]     Loss: 0.642680  Accuracy: 0.84
Train Epoch: 2 [32000/60000 (53%)]     Loss: 0.567285  Accuracy: 0.86
Train Epoch: 2 [38400/60000 (64%)]     Loss: 0.642652  Accuracy: 0.83
Train Epoch: 2 [44800/60000 (75%)]     Loss: 0.427164  Accuracy: 0.92
Train Epoch: 2 [51200/60000 (85%)]     Loss: 0.563865  Accuracy: 0.88
Train Epoch: 2 [57600/60000 (96%)]     Loss: 0.361532  Accuracy: 0.92

Test set: Average loss: 0.4687, Accuracy: 8857/10000 (89%)

Train Epoch: 3 [0/60000 (0%)]  Loss: 0.389454  Accuracy: 0.91
Train Epoch: 3 [6400/60000 (11%)]      Loss: 0.336665  Accuracy: 0.92
Train Epoch: 3 [12800/60000 (21%)]     Loss: 0.561202  Accuracy: 0.86
Train Epoch: 3 [19200/60000 (32%)]     Loss: 0.588665  Accuracy: 0.80
Train Epoch: 3 [25600/60000 (43%)]     Loss: 0.483497  Accuracy: 0.86
Train Epoch: 3 [32000/60000 (53%)]     Loss: 0.263619  Accuracy: 0.97
Train Epoch: 3 [38400/60000 (64%)]     Loss: 0.547468  Accuracy: 0.86
Train Epoch: 3 [44800/60000 (75%)]     Loss: 0.432945  Accuracy: 0.89
Train Epoch: 3 [51200/60000 (85%)]     Loss: 0.410946  Accuracy: 0.89
Train Epoch: 3 [57600/60000 (96%)]     Loss: 0.635896  Accuracy: 0.84
```

(continues on next page)

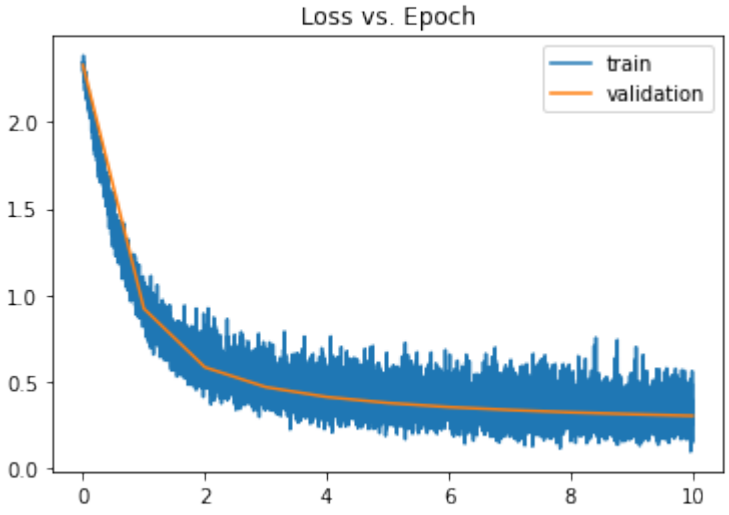
(continued from previous page)

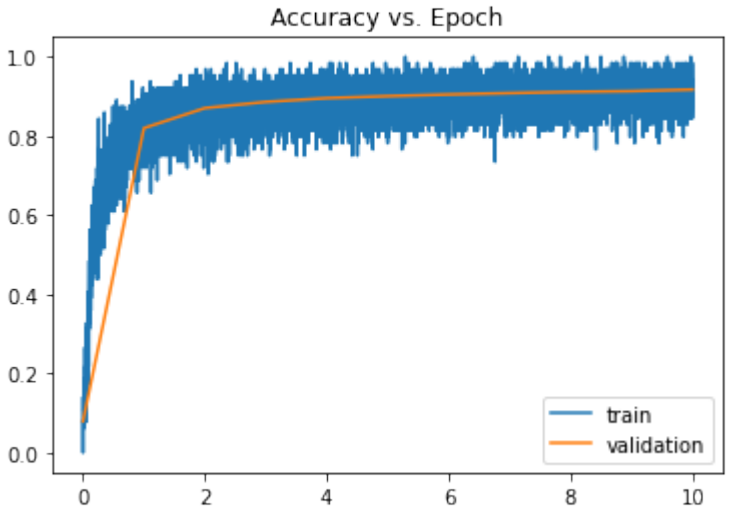
```
Test set: Average loss: 0.4115, Accuracy: 8952/10000 (90%)

Train Epoch: 4 [0/60000 (0%)]  Loss: 0.297894  Accuracy: 0.97
Train Epoch: 4 [6400/60000 (11%)]      Loss: 0.463711  Accuracy: 0.88
Train Epoch: 4 [12800/60000 (21%)]     Loss: 0.372766  Accuracy: 0.89
Train Epoch: 4 [19200/60000 (32%)]     Loss: 0.444618  Accuracy: 0.86
Train Epoch: 4 [25600/60000 (43%)]     Loss: 0.448341  Accuracy: 0.89
Train Epoch: 4 [32000/60000 (53%)]     Loss: 0.381806  Accuracy: 0.89
Train Epoch: 4 [38400/60000 (64%)]     Loss: 0.401021  Accuracy: 0.91
Train Epoch: 4 [44800/60000 (75%)]     Loss: 0.291500  Accuracy: 0.94
Train Epoch: 4 [51200/60000 (85%)]     Loss: 0.448159  Accuracy: 0.88
Train Epoch: 4 [57600/60000 (96%)]     Loss: 0.363260  Accuracy: 0.91

Test set: Average loss: 0.3770, Accuracy: 9003/10000 (90%)

Train Epoch: 5 [0/60000 (0%)]  Loss: 0.480067  Accuracy: 0.80
Train Epoch: 5 [6400/60000 (11%)]      Loss: 0.371839  Accuracy: 0.89
Train Epoch: 5 [12800/60000 (21%)]     Loss: 0.308018  Accuracy: 0.88
Train Epoch: 5 [19200/60000 (32%)]     Loss: 0.479781  Accuracy: 0.88
Train Epoch: 5 [25600/60000 (43%)]     Loss: 0.462275  Accuracy: 0.88
Train Epoch: 5 [32000/60000 (53%)]     Loss: 0.521780  Accuracy: 0.88
Train Epoch: 5 [38400/60000 (64%)]     Loss: 0.412820  Accuracy: 0.94
Train Epoch: 5 [44800/60000 (75%)]     Loss: 0.457761  Accuracy: 0.88
Train Epoch: 5 [51200/60000 (85%)]     Loss: 0.285547  Accuracy: 0.91
Train Epoch: 5 [57600/60000 (96%)]     Loss: 0.283531  Accuracy: 0.92

Test set: Average loss: 0.3532, Accuracy: 9045/10000 (90%)

Train Epoch: 6 [0/60000 (0%)]  Loss: 0.344598  Accuracy: 0.92
Train Epoch: 6 [6400/60000 (11%)]      Loss: 0.442519  Accuracy: 0.86
Train Epoch: 6 [12800/60000 (21%)]     Loss: 0.262154  Accuracy: 0.89
Train Epoch: 6 [19200/60000 (32%)]     Loss: 0.483225  Accuracy: 0.88
Train Epoch: 6 [25600/60000 (43%)]     Loss: 0.438983  Accuracy: 0.92
Train Epoch: 6 [32000/60000 (53%)]     Loss: 0.261459  Accuracy: 0.94
Train Epoch: 6 [38400/60000 (64%)]     Loss: 0.284205  Accuracy: 0.91
Train Epoch: 6 [44800/60000 (75%)]     Loss: 0.252227  Accuracy: 0.91
Train Epoch: 6 [51200/60000 (85%)]     Loss: 0.363230  Accuracy: 0.91
Train Epoch: 6 [57600/60000 (96%)]     Loss: 0.299433  Accuracy: 0.92

Test set: Average loss: 0.3365, Accuracy: 9079/10000 (91%)

Train Epoch: 7 [0/60000 (0%)]  Loss: 0.428603  Accuracy: 0.86
Train Epoch: 7 [6400/60000 (11%)]      Loss: 0.399156  Accuracy: 0.91
Train Epoch: 7 [12800/60000 (21%)]     Loss: 0.236607  Accuracy: 0.95
Train Epoch: 7 [19200/60000 (32%)]     Loss: 0.249380  Accuracy: 0.94
Train Epoch: 7 [25600/60000 (43%)]     Loss: 0.397530  Accuracy: 0.84
Train Epoch: 7 [32000/60000 (53%)]     Loss: 0.336956  Accuracy: 0.88
Train Epoch: 7 [38400/60000 (64%)]     Loss: 0.278071  Accuracy: 0.89
Train Epoch: 7 [44800/60000 (75%)]     Loss: 0.602772  Accuracy: 0.84
Train Epoch: 7 [51200/60000 (85%)]     Loss: 0.351761  Accuracy: 0.91
Train Epoch: 7 [57600/60000 (96%)]     Loss: 0.467826  Accuracy: 0.88
```

(continues on next page)

(continued from previous page)

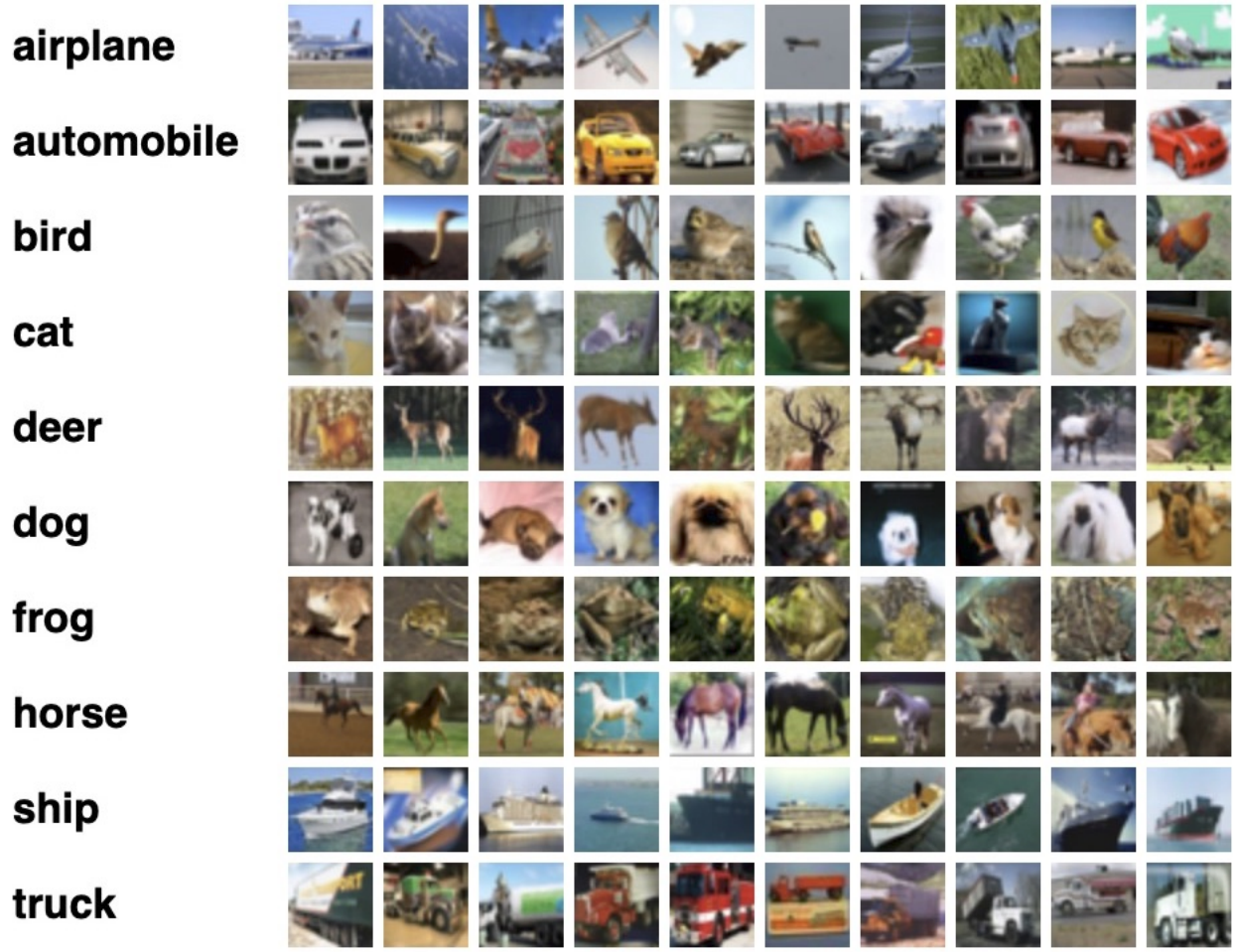
```
Test set: Average loss: 0.3226, Accuracy: 9110/10000 (91%)

Train Epoch: 8 [0/60000 (0%)]  Loss: 0.508125  Accuracy: 0.91
Train Epoch: 8 [6400/60000 (11%)]      Loss: 0.336828  Accuracy: 0.91
Train Epoch: 8 [12800/60000 (21%)]     Loss: 0.514461  Accuracy: 0.88
Train Epoch: 8 [19200/60000 (32%)]     Loss: 0.369944  Accuracy: 0.86
Train Epoch: 8 [25600/60000 (43%)]     Loss: 0.290642  Accuracy: 0.92
Train Epoch: 8 [32000/60000 (53%)]     Loss: 0.240326  Accuracy: 0.97
Train Epoch: 8 [38400/60000 (64%)]     Loss: 0.319198  Accuracy: 0.94
Train Epoch: 8 [44800/60000 (75%)]     Loss: 0.381088  Accuracy: 0.88
Train Epoch: 8 [51200/60000 (85%)]     Loss: 0.390811  Accuracy: 0.94
Train Epoch: 8 [57600/60000 (96%)]     Loss: 0.487675  Accuracy: 0.84

Test set: Average loss: 0.3118, Accuracy: 9128/10000 (91%)

Train Epoch: 9 [0/60000 (0%)]  Loss: 0.426937  Accuracy: 0.86
Train Epoch: 9 [6400/60000 (11%)]      Loss: 0.327623  Accuracy: 0.89
Train Epoch: 9 [12800/60000 (21%)]     Loss: 0.178158  Accuracy: 0.95
Train Epoch: 9 [19200/60000 (32%)]     Loss: 0.254056  Accuracy: 0.92
Train Epoch: 9 [25600/60000 (43%)]     Loss: 0.270350  Accuracy: 0.92
Train Epoch: 9 [32000/60000 (53%)]     Loss: 0.326204  Accuracy: 0.95
Train Epoch: 9 [38400/60000 (64%)]     Loss: 0.238152  Accuracy: 0.95
Train Epoch: 9 [44800/60000 (75%)]     Loss: 0.231811  Accuracy: 0.94
Train Epoch: 9 [51200/60000 (85%)]     Loss: 0.326061  Accuracy: 0.89
Train Epoch: 9 [57600/60000 (96%)]     Loss: 0.170712  Accuracy: 0.95

Test set: Average loss: 0.3024, Accuracy: 9169/10000 (92%)
```





2.18 What about harder datasets?

MNIST is the easiest widely-used dataset as a computer vision toy-problem. One step above MNIST is CIFAR10, which contains images from 10 classes of objects:



[15]:
```
BATCH_SIZE = 64
EPOCHS = 10
LEARNING_RATE = 0.01
LOG_INTERVAL = 100

model = FCNN(input_size=32*32*3).to(device)
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalize the data to 0 mean and 1 standard deviation, now for all channels of RGB
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

train_loader, test_loader = (
    torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=train, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=True,
        shuffle=train
    )
    for train in (True, False)
```

(continues on next page)

(continued from previous page)

```
)

fit(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_INTERVAL)
```

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

Test set: Average loss: 2.3326, Accuracy: 1252/10000 (13%)

Train Epoch: 0 [0/50000 (0%)]  Loss: 2.300806  Accuracy: 0.14
Train Epoch: 0 [6400/50000 (13%)]      Loss: 1.851318  Accuracy: 0.38
Train Epoch: 0 [12800/50000 (26%)]     Loss: 1.687283  Accuracy: 0.44
Train Epoch: 0 [19200/50000 (38%)]     Loss: 1.610826  Accuracy: 0.44
Train Epoch: 0 [25600/50000 (51%)]     Loss: 1.610463  Accuracy: 0.48
Train Epoch: 0 [32000/50000 (64%)]     Loss: 1.740376  Accuracy: 0.42
Train Epoch: 0 [38400/50000 (77%)]     Loss: 1.630639  Accuracy: 0.41
Train Epoch: 0 [44800/50000 (90%)]     Loss: 1.620754  Accuracy: 0.48

Test set: Average loss: 1.5749, Accuracy: 4445/10000 (44%)

Train Epoch: 1 [0/50000 (0%)]  Loss: 1.653425  Accuracy: 0.41
Train Epoch: 1 [6400/50000 (13%)]      Loss: 1.531922  Accuracy: 0.50
Train Epoch: 1 [12800/50000 (26%)]     Loss: 1.735946  Accuracy: 0.36
Train Epoch: 1 [19200/50000 (38%)]     Loss: 1.510094  Accuracy: 0.47
Train Epoch: 1 [25600/50000 (51%)]     Loss: 1.537139  Accuracy: 0.42
Train Epoch: 1 [32000/50000 (64%)]     Loss: 1.528880  Accuracy: 0.44
Train Epoch: 1 [38400/50000 (77%)]     Loss: 1.407133  Accuracy: 0.52
Train Epoch: 1 [44800/50000 (90%)]     Loss: 1.526339  Accuracy: 0.56

Test set: Average loss: 1.4859, Accuracy: 4766/10000 (48%)

Train Epoch: 2 [0/50000 (0%)]  Loss: 1.317183  Accuracy: 0.55
Train Epoch: 2 [6400/50000 (13%)]      Loss: 1.399197  Accuracy: 0.48
Train Epoch: 2 [12800/50000 (26%)]     Loss: 1.516985  Accuracy: 0.36
Train Epoch: 2 [19200/50000 (38%)]     Loss: 1.383561  Accuracy: 0.50
Train Epoch: 2 [25600/50000 (51%)]     Loss: 1.298355  Accuracy: 0.56
Train Epoch: 2 [32000/50000 (64%)]     Loss: 1.497987  Accuracy: 0.50
Train Epoch: 2 [38400/50000 (77%)]     Loss: 1.515899  Accuracy: 0.53
Train Epoch: 2 [44800/50000 (90%)]     Loss: 1.568963  Accuracy: 0.45

Test set: Average loss: 1.4556, Accuracy: 4865/10000 (49%)

Train Epoch: 3 [0/50000 (0%)]  Loss: 1.312663  Accuracy: 0.55
Train Epoch: 3 [6400/50000 (13%)]      Loss: 1.323932  Accuracy: 0.55
Train Epoch: 3 [12800/50000 (26%)]     Loss: 1.340122  Accuracy: 0.56
Train Epoch: 3 [19200/50000 (38%)]     Loss: 1.312385  Accuracy: 0.59
Train Epoch: 3 [25600/50000 (51%)]     Loss: 1.192983  Accuracy: 0.66
Train Epoch: 3 [32000/50000 (64%)]     Loss: 1.313574  Accuracy: 0.55
Train Epoch: 3 [38400/50000 (77%)]     Loss: 1.301565  Accuracy: 0.59
Train Epoch: 3 [44800/50000 (90%)]     Loss: 1.378752  Accuracy: 0.50

Test set: Average loss: 1.4190, Accuracy: 5019/10000 (50%)
```

(continues on next page)

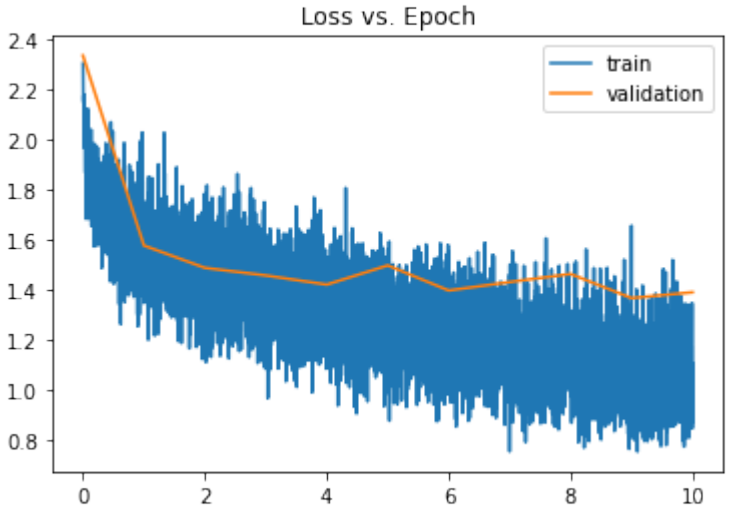
(continued from previous page)

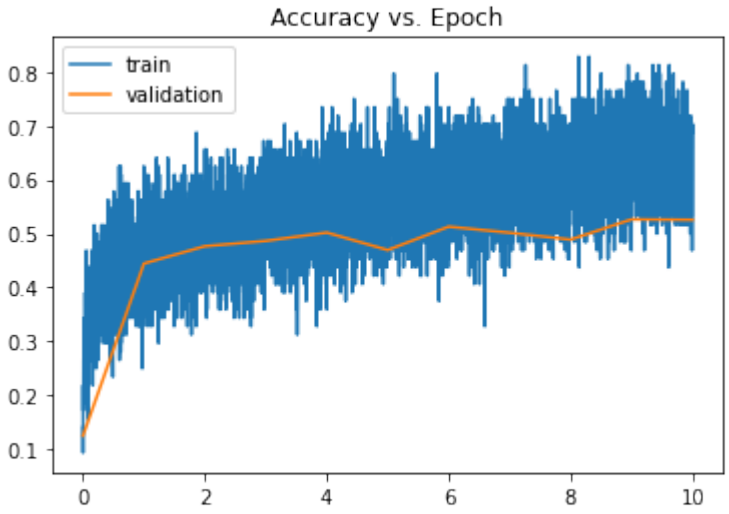
```
Train Epoch: 4 [0/50000 (0%)]  Loss: 1.383258  Accuracy: 0.50
Train Epoch: 4 [6400/50000 (13%)]      Loss: 1.475597  Accuracy: 0.45
Train Epoch: 4 [12800/50000 (26%)]     Loss: 1.102977  Accuracy: 0.64
Train Epoch: 4 [19200/50000 (38%)]     Loss: 1.452052  Accuracy: 0.48
Train Epoch: 4 [25600/50000 (51%)]     Loss: 1.465976  Accuracy: 0.50
Train Epoch: 4 [32000/50000 (64%)]     Loss: 1.265100  Accuracy: 0.58
Train Epoch: 4 [38400/50000 (77%)]     Loss: 1.300325  Accuracy: 0.50
Train Epoch: 4 [44800/50000 (90%)]     Loss: 1.209848  Accuracy: 0.56

Test set: Average loss: 1.4955, Accuracy: 4695/10000 (47%)

Train Epoch: 5 [0/50000 (0%)]  Loss: 1.561614  Accuracy: 0.47
Train Epoch: 5 [6400/50000 (13%)]      Loss: 1.389365  Accuracy: 0.52
Train Epoch: 5 [12800/50000 (26%)]     Loss: 1.277032  Accuracy: 0.55
Train Epoch: 5 [19200/50000 (38%)]     Loss: 1.204583  Accuracy: 0.59
Train Epoch: 5 [25600/50000 (51%)]     Loss: 1.220117  Accuracy: 0.64
Train Epoch: 5 [32000/50000 (64%)]     Loss: 1.189834  Accuracy: 0.59
Train Epoch: 5 [38400/50000 (77%)]     Loss: 1.185191  Accuracy: 0.59
Train Epoch: 5 [44800/50000 (90%)]     Loss: 1.220989  Accuracy: 0.62

Test set: Average loss: 1.3955, Accuracy: 5129/10000 (51%)

Train Epoch: 6 [0/50000 (0%)]  Loss: 1.217850  Accuracy: 0.61
Train Epoch: 6 [6400/50000 (13%)]      Loss: 1.379431  Accuracy: 0.50
Train Epoch: 6 [12800/50000 (26%)]     Loss: 1.000018  Accuracy: 0.64
Train Epoch: 6 [19200/50000 (38%)]     Loss: 1.378863  Accuracy: 0.55
Train Epoch: 6 [25600/50000 (51%)]     Loss: 1.247362  Accuracy: 0.59
Train Epoch: 6 [32000/50000 (64%)]     Loss: 1.139954  Accuracy: 0.59
Train Epoch: 6 [38400/50000 (77%)]     Loss: 1.142104  Accuracy: 0.62
Train Epoch: 6 [44800/50000 (90%)]     Loss: 1.294392  Accuracy: 0.59

Test set: Average loss: 1.4291, Accuracy: 5016/10000 (50%)

Train Epoch: 7 [0/50000 (0%)]  Loss: 1.104913  Accuracy: 0.62
Train Epoch: 7 [6400/50000 (13%)]      Loss: 0.964932  Accuracy: 0.67
Train Epoch: 7 [12800/50000 (26%)]     Loss: 1.112383  Accuracy: 0.67
Train Epoch: 7 [19200/50000 (38%)]     Loss: 1.278114  Accuracy: 0.55
Train Epoch: 7 [25600/50000 (51%)]     Loss: 1.377846  Accuracy: 0.45
Train Epoch: 7 [32000/50000 (64%)]     Loss: 1.104867  Accuracy: 0.58
Train Epoch: 7 [38400/50000 (77%)]     Loss: 1.024559  Accuracy: 0.69
Train Epoch: 7 [44800/50000 (90%)]     Loss: 1.220774  Accuracy: 0.55

Test set: Average loss: 1.4614, Accuracy: 4889/10000 (49%)

Train Epoch: 8 [0/50000 (0%)]  Loss: 1.258132  Accuracy: 0.55
Train Epoch: 8 [6400/50000 (13%)]      Loss: 0.789627  Accuracy: 0.83
Train Epoch: 8 [12800/50000 (26%)]     Loss: 1.265380  Accuracy: 0.55
Train Epoch: 8 [19200/50000 (38%)]     Loss: 1.008643  Accuracy: 0.62
Train Epoch: 8 [25600/50000 (51%)]     Loss: 1.144870  Accuracy: 0.66
Train Epoch: 8 [32000/50000 (64%)]     Loss: 1.044951  Accuracy: 0.67
Train Epoch: 8 [38400/50000 (77%)]     Loss: 1.105204  Accuracy: 0.62
```

(continues on next page)

(continued from previous page)

```
Train Epoch: 8 [44800/50000 (90%)]     Loss: 1.209067  Accuracy: 0.59

Test set: Average loss: 1.3650, Accuracy: 5264/10000 (53%)

Train Epoch: 9 [0/50000 (0%)]  Loss: 1.094587  Accuracy: 0.66
Train Epoch: 9 [6400/50000 (13%)]      Loss: 1.030614  Accuracy: 0.59
Train Epoch: 9 [12800/50000 (26%)]     Loss: 1.061698  Accuracy: 0.58
Train Epoch: 9 [19200/50000 (38%)]     Loss: 0.898910  Accuracy: 0.80
Train Epoch: 9 [25600/50000 (51%)]     Loss: 0.975828  Accuracy: 0.62
Train Epoch: 9 [32000/50000 (64%)]     Loss: 1.281817  Accuracy: 0.58
Train Epoch: 9 [38400/50000 (77%)]     Loss: 0.986419  Accuracy: 0.64
Train Epoch: 9 [44800/50000 (90%)]     Loss: 1.232195  Accuracy: 0.56

Test set: Average loss: 1.3883, Accuracy: 5256/10000 (53%)
```





2.19 Ai, probably not the 90%+ accuracy we saw with MNIST!

What can you do about this? Try something out! For example: - Bigger model - Different optimizer - Different learning rate - More epochs - Data augmentation

Each methods has its upsides and downsides, think about what these are before changing something!

But maybe we can change the model to incorporate information about the data before even starting learning… ### Next chapter: CNNs; improving Fully-Connected networks with prior knowledge

[8]:
```
import os
from typing import Sequence, Tuple
from datetime import datetime

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchmetrics.functional as metrics
import numpy as np
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

%matplotlib inline

DATA_PATH = os.getenv('TEACHER_DIR', os.getcwd()) + '/JHL_data'
```

2.20 Introducing Convolution! What is it?

Before, we built a network that accepts the normalized pixel values of each value and operates soley on those values. What if we could instead feed different features (e.g. curvature, edges) of each image into a network, and have the network learn which features are important for classifying an image?

This possible through convolution! Convolution applies kernels (filters) that traverse through each image and generate feature maps, see:

https://miro.medium.com/max/500/1*GcI7G-JLAQiEoCON7xFbhg.gif

In the above example, the image is a 5 x 5 matrix and the kernel going over it is a 3 x 3 matrix. A dot product operation takes place between the image and the kernel and the convolved feature is generated. Each kernel in a CNN learns a different characteristic of an image.

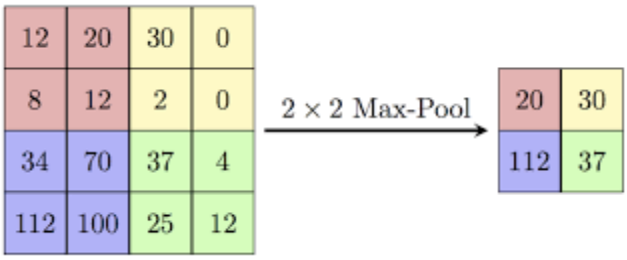
2.20.1 In the olden days:

Kernels are often used in photoediting software to apply blurring, edge detection, sharpening, etc.

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

2.20.2 Now:

Kernels in deep learning networks are used in similar ways, i.e. highlighting some feature. Combined with a system called max pooling, the non-highlighted elements are discarded from each feature map, leaving only the features of interest, reducing the number of learned parameters, and decreasing the computational cost (e.g. system memory).



We can also take convolutions of convolutions – we can stack as many convolutions as we want, as long as there are enough pixels to fit a kernel.

Warning: What you may find down there in those deep convolutions may not appear recognizable to you, see also: https://distill.pub/2019/activation-atlas/

2.21 How to continue?

We will try to classify CIFAR10 with our own CNN, re-using our train/test loops

[9]:
```
def train(model, device, train_loader, optimizer, epoch, log_interval=10):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        # move data and target to the gpu, if available and used
        data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
↪target))

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
```

(continues on next page)

(continued from previous page)

```
        optimizer.step()

        accuracy = metrics.accuracy(output, target)

        if batch_idx % log_interval == 0:
            print(
                f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                f'\tLoss: {loss.detach().item():.6f}'
                f'\tAccuracy: {accuracy.detach().item():.2f}'
            )

        yield loss.detach().item(), accuracy.detach().item()


@torch.no_grad()
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0

    for data, target in test_loader:
        # move data and target to the gpu, if available and used
        data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
↪target))

        # get model output
        output = model(data)

        # calculate loss
        test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch␣
↪loss

        # get most likely class label
        pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-
↪probability

        # count the number of correct predictions
        correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100 * correct / len(test_loader.dataset)))

    yield test_loss, correct / len(test_loader.dataset)
```

[10]:
```
def plot_metric_curve(
    train_metric: Sequence[float],
    val_metric: Sequence[float],
    n_epochs: int,
```

(continues on next page)

(continued from previous page)

```
    metric_name: str, # Label of the y-axis, e.g. 'Accuracy'
    x_axis_name: str = 'Epoch'
):
    # create values for the x-axis
    train_steps, val_steps = map(
        lambda metric_values: np.linspace(start=0, stop=n_epochs, num=len(metric_
↪values)),
        (train_metric, val_metric)
    )

    plt.plot(train_steps, train_metric, label='train')
    plt.plot(val_steps, val_metric, label='validation')
    plt.title(f"{metric_name} vs. {x_axis_name}")
    plt.legend()
    plt.show()
```

[11]:
```
def fit(model, optimizer, n_epochs, device, train_loader, test_loader, log_interval):

    # get the validation loss and accuracy of the untrained model
    start_val_loss, start_val_acc = tuple(test(model, device, test_loader))[0]

    # don't mind the following train/test loop logic too much, if you want to know what's␣
↪happening, let us know :)
    # normally you would pass a logger to your train/test loops and log the respective␣
↪metrics there
    (train_loss, train_acc), (val_loss, val_acc) = map(lambda arr: np.asarray(arr).
↪transpose(2,0,1), zip(*[
        (
            [*train(model, device, train_loader, optimizer, epoch, log_interval)],
            [*test(model, device, test_loader)]
        )
        for epoch in range(n_epochs)
    ]))

    # flatten the arrays
    train_loss, train_acc, val_loss, val_acc = map(np.ravel, (train_loss, train_acc, val_
↪loss, val_acc))

    # prepend the validation loss and accuracy of the untrained model
    val_loss, val_acc = (start_val_loss, *val_loss), (start_val_acc, *val_acc)

    plot_metric_curve(train_loss, val_loss, n_epochs, 'Loss')
    plot_metric_curve(train_acc, val_acc, n_epochs, 'Accuracy')
```

2.22 Defining the model

Moving from a FCNN to a CNN, we split our model in two: a feature extractor and a classifier.
The classifier will be no different to the FCNN: Some linear layers with a non-linearity in-between.
The feature extractor will be our convolutional layers, with a downsampling operation after each layer; in our case max-pooling.
(Which you shouldn't normally use unless you have a very good reason to! Take it from Saint Geoff: https://mirror2image.wordpress.com/2014/11/11/geoffrey-hinton-on-max-pooling-reddit-ama/))

[12]:
```
class CIFAR10CNN(nn.Module):
    def __init__(self):
        super().__init__()

        # 4 convolution layers, with a non-linear activation after each.
        # maxpooling after the activations of the 2nd, 3rd, and 4th conv layers
        # 2 dense layers for classification
        # log_softmax
        #
        # As for the number of channels of each layers, try to experiment!

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, padding=1),
            nn.ReLU(),

            nn.Conv2d(in_channels=8, out_channels=32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )

        # in_features of the first layer should be the product of the output shape of␣
↪your feature extractor!
        # E.g. if the output of your feature extractor has size (batch x 128 x 4 x 4),␣
↪in_features = 128*4*4=2048
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=128*4*4, out_features=4096),
            nn.ReLU(),
            nn.Linear(in_features=4096, out_features=10),
            nn.LogSoftmax(dim=1)
        )
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        features = self.feature_extractor(x)

        return self.classifier(features)
```

[13]:
```
use_cuda = torch.cuda.is_available()
print(f"CUDA is {'' if use_cuda else 'not '}available")
device = torch.device("cuda" if use_cuda else "cpu")

if use_cuda:
    torch.cuda.set_per_process_memory_fraction(0.22)

cpu_count = len(os.sched_getaffinity(0))
```

```
CUDA is available
```

[14]:
```
# We can Re-use our train and test loops

BATCH_SIZE = 600
EPOCHS = 20
LEARNING_RATE = 1e-4

LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

model = CIFAR10CNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalize the data to 0 mean and 1 standard deviation, now for all channels of RGB
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

train_loader, test_loader = (
    torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=train, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=train,
        num_workers=cpu_count
    )
    for train in (True, False)
)

fit(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_INTERVAL)
```

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

Test set: Average loss: 2.3027, Accuracy: 883/10000 (9%)
```

(continues on next page)

(continued from previous page)

```
Train Epoch: 0 [0/50000 (0%)]  Loss: 2.302135  Accuracy: 0.08
Train Epoch: 0 [6000/50000 (12%)]      Loss: 2.234964  Accuracy: 0.20
Train Epoch: 0 [12000/50000 (24%)]     Loss: 2.134983  Accuracy: 0.26
Train Epoch: 0 [18000/50000 (36%)]     Loss: 2.060714  Accuracy: 0.30
Train Epoch: 0 [24000/50000 (48%)]     Loss: 1.946920  Accuracy: 0.32
Train Epoch: 0 [30000/50000 (60%)]     Loss: 1.838532  Accuracy: 0.32
Train Epoch: 0 [36000/50000 (71%)]     Loss: 1.816722  Accuracy: 0.37
Train Epoch: 0 [42000/50000 (83%)]     Loss: 1.817462  Accuracy: 0.34
Train Epoch: 0 [48000/50000 (95%)]     Loss: 1.756518  Accuracy: 0.37

Test set: Average loss: 1.7060, Accuracy: 3966/10000 (40%)

Train Epoch: 1 [0/50000 (0%)]  Loss: 1.693645  Accuracy: 0.40
Train Epoch: 1 [6000/50000 (12%)]      Loss: 1.731694  Accuracy: 0.36
Train Epoch: 1 [12000/50000 (24%)]     Loss: 1.635071  Accuracy: 0.40
Train Epoch: 1 [18000/50000 (36%)]     Loss: 1.620123  Accuracy: 0.42
Train Epoch: 1 [24000/50000 (48%)]     Loss: 1.653806  Accuracy: 0.38
Train Epoch: 1 [30000/50000 (60%)]     Loss: 1.579616  Accuracy: 0.44
Train Epoch: 1 [36000/50000 (71%)]     Loss: 1.593030  Accuracy: 0.42
Train Epoch: 1 [42000/50000 (83%)]     Loss: 1.617004  Accuracy: 0.43
Train Epoch: 1 [48000/50000 (95%)]     Loss: 1.514681  Accuracy: 0.48

Test set: Average loss: 1.5281, Accuracy: 4605/10000 (46%)

Train Epoch: 2 [0/50000 (0%)]  Loss: 1.560438  Accuracy: 0.44
Train Epoch: 2 [6000/50000 (12%)]      Loss: 1.530639  Accuracy: 0.48
Train Epoch: 2 [12000/50000 (24%)]     Loss: 1.519998  Accuracy: 0.46
Train Epoch: 2 [18000/50000 (36%)]     Loss: 1.478619  Accuracy: 0.48
Train Epoch: 2 [24000/50000 (48%)]     Loss: 1.492016  Accuracy: 0.45
Train Epoch: 2 [30000/50000 (60%)]     Loss: 1.457478  Accuracy: 0.48
Train Epoch: 2 [36000/50000 (71%)]     Loss: 1.485044  Accuracy: 0.44
Train Epoch: 2 [42000/50000 (83%)]     Loss: 1.453706  Accuracy: 0.47
Train Epoch: 2 [48000/50000 (95%)]     Loss: 1.445858  Accuracy: 0.50

Test set: Average loss: 1.4291, Accuracy: 4864/10000 (49%)

Train Epoch: 3 [0/50000 (0%)]  Loss: 1.461212  Accuracy: 0.52
Train Epoch: 3 [6000/50000 (12%)]      Loss: 1.374049  Accuracy: 0.52
Train Epoch: 3 [12000/50000 (24%)]     Loss: 1.517243  Accuracy: 0.45
Train Epoch: 3 [18000/50000 (36%)]     Loss: 1.409744  Accuracy: 0.50
Train Epoch: 3 [24000/50000 (48%)]     Loss: 1.331850  Accuracy: 0.53
Train Epoch: 3 [30000/50000 (60%)]     Loss: 1.407649  Accuracy: 0.51
Train Epoch: 3 [36000/50000 (71%)]     Loss: 1.368647  Accuracy: 0.49
Train Epoch: 3 [42000/50000 (83%)]     Loss: 1.349765  Accuracy: 0.53
Train Epoch: 3 [48000/50000 (95%)]     Loss: 1.373083  Accuracy: 0.52

Test set: Average loss: 1.3846, Accuracy: 5074/10000 (51%)

Train Epoch: 4 [0/50000 (0%)]  Loss: 1.358583  Accuracy: 0.51
Train Epoch: 4 [6000/50000 (12%)]      Loss: 1.404799  Accuracy: 0.51
Train Epoch: 4 [12000/50000 (24%)]     Loss: 1.397840  Accuracy: 0.53
```

(continues on next page)

(continued from previous page)

```
Train Epoch: 4 [18000/50000 (36%)]     Loss: 1.424757  Accuracy: 0.48
Train Epoch: 4 [24000/50000 (48%)]     Loss: 1.324732  Accuracy: 0.53
Train Epoch: 4 [30000/50000 (60%)]     Loss: 1.341245  Accuracy: 0.55
Train Epoch: 4 [36000/50000 (71%)]     Loss: 1.356028  Accuracy: 0.49
Train Epoch: 4 [42000/50000 (83%)]     Loss: 1.332021  Accuracy: 0.51
Train Epoch: 4 [48000/50000 (95%)]     Loss: 1.263047  Accuracy: 0.54

Test set: Average loss: 1.3349, Accuracy: 5211/10000 (52%)

Train Epoch: 5 [0/50000 (0%)]  Loss: 1.306721  Accuracy: 0.54
Train Epoch: 5 [6000/50000 (12%)]      Loss: 1.284430  Accuracy: 0.55
Train Epoch: 5 [12000/50000 (24%)]     Loss: 1.350179  Accuracy: 0.52
Train Epoch: 5 [18000/50000 (36%)]     Loss: 1.268469  Accuracy: 0.55
Train Epoch: 5 [24000/50000 (48%)]     Loss: 1.374626  Accuracy: 0.51
Train Epoch: 5 [30000/50000 (60%)]     Loss: 1.279148  Accuracy: 0.53
Train Epoch: 5 [36000/50000 (71%)]     Loss: 1.365725  Accuracy: 0.53
Train Epoch: 5 [42000/50000 (83%)]     Loss: 1.411205  Accuracy: 0.48
Train Epoch: 5 [48000/50000 (95%)]     Loss: 1.286952  Accuracy: 0.55

Test set: Average loss: 1.3031, Accuracy: 5305/10000 (53%)

Train Epoch: 6 [0/50000 (0%)]  Loss: 1.299689  Accuracy: 0.51
Train Epoch: 6 [6000/50000 (12%)]      Loss: 1.279058  Accuracy: 0.54
Train Epoch: 6 [12000/50000 (24%)]     Loss: 1.337447  Accuracy: 0.53
Train Epoch: 6 [18000/50000 (36%)]     Loss: 1.339325  Accuracy: 0.52
Train Epoch: 6 [24000/50000 (48%)]     Loss: 1.218355  Accuracy: 0.55
Train Epoch: 6 [30000/50000 (60%)]     Loss: 1.197334  Accuracy: 0.58
Train Epoch: 6 [36000/50000 (71%)]     Loss: 1.246907  Accuracy: 0.58
Train Epoch: 6 [42000/50000 (83%)]     Loss: 1.203366  Accuracy: 0.56
Train Epoch: 6 [48000/50000 (95%)]     Loss: 1.276328  Accuracy: 0.54

Test set: Average loss: 1.2615, Accuracy: 5489/10000 (55%)

Train Epoch: 7 [0/50000 (0%)]  Loss: 1.223782  Accuracy: 0.58
Train Epoch: 7 [6000/50000 (12%)]      Loss: 1.255562  Accuracy: 0.57
Train Epoch: 7 [12000/50000 (24%)]     Loss: 1.209092  Accuracy: 0.56
Train Epoch: 7 [18000/50000 (36%)]     Loss: 1.285838  Accuracy: 0.54
Train Epoch: 7 [24000/50000 (48%)]     Loss: 1.344548  Accuracy: 0.52
Train Epoch: 7 [30000/50000 (60%)]     Loss: 1.206154  Accuracy: 0.53
Train Epoch: 7 [36000/50000 (71%)]     Loss: 1.218868  Accuracy: 0.55
Train Epoch: 7 [42000/50000 (83%)]     Loss: 1.251492  Accuracy: 0.56
Train Epoch: 7 [48000/50000 (95%)]     Loss: 1.191027  Accuracy: 0.56

Test set: Average loss: 1.2435, Accuracy: 5619/10000 (56%)

Train Epoch: 8 [0/50000 (0%)]  Loss: 1.140604  Accuracy: 0.60
Train Epoch: 8 [6000/50000 (12%)]      Loss: 1.208094  Accuracy: 0.57
Train Epoch: 8 [12000/50000 (24%)]     Loss: 1.209153  Accuracy: 0.60
Train Epoch: 8 [18000/50000 (36%)]     Loss: 1.252662  Accuracy: 0.54
Train Epoch: 8 [24000/50000 (48%)]     Loss: 1.146931  Accuracy: 0.61
Train Epoch: 8 [30000/50000 (60%)]     Loss: 1.214656  Accuracy: 0.58
Train Epoch: 8 [36000/50000 (71%)]     Loss: 1.189945  Accuracy: 0.60
```

(continues on next page)

(continued from previous page)

```
Train Epoch: 8 [42000/50000 (83%)]     Loss: 1.257438  Accuracy: 0.57
Train Epoch: 8 [48000/50000 (95%)]     Loss: 1.222701  Accuracy: 0.58

Test set: Average loss: 1.2005, Accuracy: 5713/10000 (57%)

Train Epoch: 9 [0/50000 (0%)]  Loss: 1.168598  Accuracy: 0.56
Train Epoch: 9 [6000/50000 (12%)]      Loss: 1.212918  Accuracy: 0.56
Train Epoch: 9 [12000/50000 (24%)]     Loss: 1.183012  Accuracy: 0.56
Train Epoch: 9 [18000/50000 (36%)]     Loss: 1.098007  Accuracy: 0.60
Train Epoch: 9 [24000/50000 (48%)]     Loss: 1.239702  Accuracy: 0.55
Train Epoch: 9 [30000/50000 (60%)]     Loss: 1.141495  Accuracy: 0.58
Train Epoch: 9 [36000/50000 (71%)]     Loss: 1.197177  Accuracy: 0.56
Train Epoch: 9 [42000/50000 (83%)]     Loss: 1.163574  Accuracy: 0.60
Train Epoch: 9 [48000/50000 (95%)]     Loss: 1.209865  Accuracy: 0.57

Test set: Average loss: 1.2005, Accuracy: 5707/10000 (57%)

Train Epoch: 10 [0/50000 (0%)] Loss: 1.125952  Accuracy: 0.60
Train Epoch: 10 [6000/50000 (12%)]     Loss: 1.230665  Accuracy: 0.58
Train Epoch: 10 [12000/50000 (24%)]    Loss: 1.142613  Accuracy: 0.62
Train Epoch: 10 [18000/50000 (36%)]    Loss: 1.159474  Accuracy: 0.59
Train Epoch: 10 [24000/50000 (48%)]    Loss: 1.129559  Accuracy: 0.60
Train Epoch: 10 [30000/50000 (60%)]    Loss: 1.151859  Accuracy: 0.60
Train Epoch: 10 [36000/50000 (71%)]    Loss: 1.134456  Accuracy: 0.59
Train Epoch: 10 [42000/50000 (83%)]    Loss: 1.128051  Accuracy: 0.60
Train Epoch: 10 [48000/50000 (95%)]    Loss: 1.150175  Accuracy: 0.58

Test set: Average loss: 1.1605, Accuracy: 5891/10000 (59%)

Train Epoch: 11 [0/50000 (0%)] Loss: 1.025830  Accuracy: 0.64
Train Epoch: 11 [6000/50000 (12%)]     Loss: 1.109209  Accuracy: 0.60
Train Epoch: 11 [12000/50000 (24%)]    Loss: 1.082912  Accuracy: 0.61
Train Epoch: 11 [18000/50000 (36%)]    Loss: 1.060199  Accuracy: 0.63
Train Epoch: 11 [24000/50000 (48%)]    Loss: 1.116448  Accuracy: 0.63
Train Epoch: 11 [30000/50000 (60%)]    Loss: 1.041413  Accuracy: 0.65
Train Epoch: 11 [36000/50000 (71%)]    Loss: 1.063143  Accuracy: 0.62
Train Epoch: 11 [42000/50000 (83%)]    Loss: 1.127012  Accuracy: 0.60
Train Epoch: 11 [48000/50000 (95%)]    Loss: 0.999790  Accuracy: 0.64

Test set: Average loss: 1.1425, Accuracy: 5885/10000 (59%)

Train Epoch: 12 [0/50000 (0%)] Loss: 1.066079  Accuracy: 0.62
Train Epoch: 12 [6000/50000 (12%)]     Loss: 1.053728  Accuracy: 0.63
Train Epoch: 12 [12000/50000 (24%)]    Loss: 1.194537  Accuracy: 0.58
Train Epoch: 12 [18000/50000 (36%)]    Loss: 1.093779  Accuracy: 0.60
Train Epoch: 12 [24000/50000 (48%)]    Loss: 0.995968  Accuracy: 0.65
Train Epoch: 12 [30000/50000 (60%)]    Loss: 1.143342  Accuracy: 0.61
Train Epoch: 12 [36000/50000 (71%)]    Loss: 1.082292  Accuracy: 0.62
Train Epoch: 12 [42000/50000 (83%)]    Loss: 1.083162  Accuracy: 0.62
Train Epoch: 12 [48000/50000 (95%)]    Loss: 1.106319  Accuracy: 0.64

Test set: Average loss: 1.1305, Accuracy: 5942/10000 (59%)
```

(continues on next page)

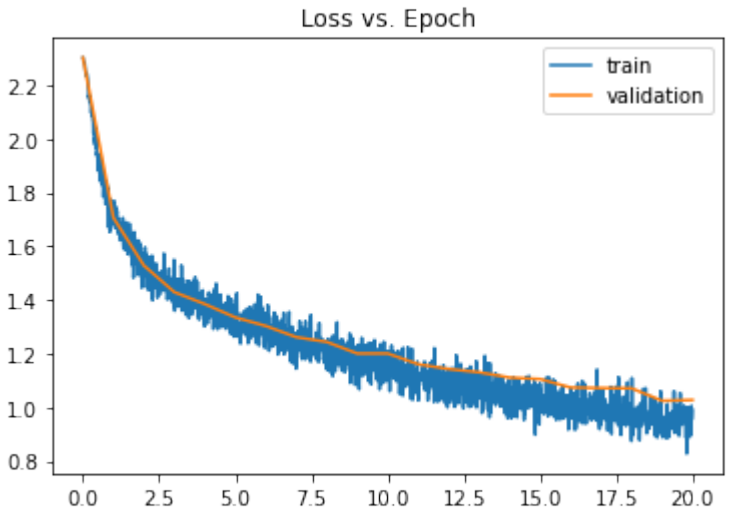
(continued from previous page)

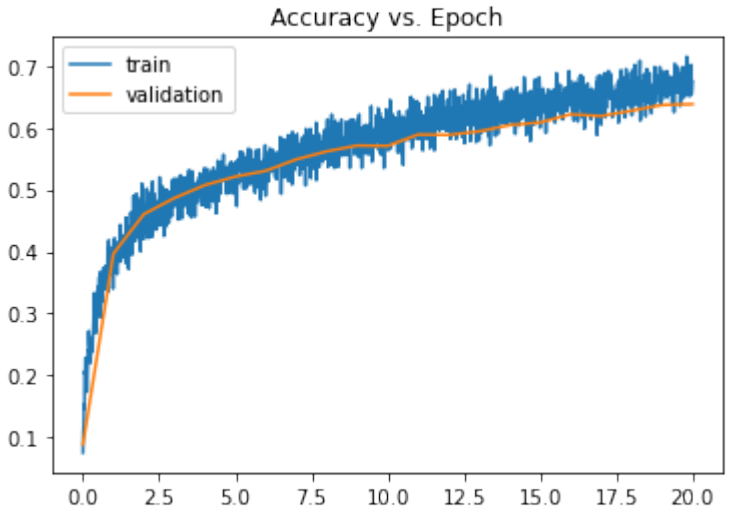
```
Train Epoch: 13 [0/50000 (0%)] Loss: 1.123214  Accuracy: 0.58
Train Epoch: 13 [6000/50000 (12%)]     Loss: 0.989526  Accuracy: 0.66
Train Epoch: 13 [12000/50000 (24%)]    Loss: 1.083638  Accuracy: 0.61
Train Epoch: 13 [18000/50000 (36%)]    Loss: 1.060837  Accuracy: 0.64
Train Epoch: 13 [24000/50000 (48%)]    Loss: 1.030054  Accuracy: 0.61
Train Epoch: 13 [30000/50000 (60%)]    Loss: 1.076957  Accuracy: 0.63
Train Epoch: 13 [36000/50000 (71%)]    Loss: 1.005130  Accuracy: 0.64
Train Epoch: 13 [42000/50000 (83%)]    Loss: 1.018066  Accuracy: 0.64
Train Epoch: 13 [48000/50000 (95%)]    Loss: 1.102728  Accuracy: 0.61

Test set: Average loss: 1.1107, Accuracy: 6043/10000 (60%)

Train Epoch: 14 [0/50000 (0%)] Loss: 1.022596  Accuracy: 0.66
Train Epoch: 14 [6000/50000 (12%)]     Loss: 1.112900  Accuracy: 0.62
Train Epoch: 14 [12000/50000 (24%)]    Loss: 0.962251  Accuracy: 0.67
Train Epoch: 14 [18000/50000 (36%)]    Loss: 1.023852  Accuracy: 0.64
Train Epoch: 14 [24000/50000 (48%)]    Loss: 1.018415  Accuracy: 0.67
Train Epoch: 14 [30000/50000 (60%)]    Loss: 1.113823  Accuracy: 0.60
Train Epoch: 14 [36000/50000 (71%)]    Loss: 1.059162  Accuracy: 0.65
Train Epoch: 14 [42000/50000 (83%)]    Loss: 0.897877  Accuracy: 0.68
Train Epoch: 14 [48000/50000 (95%)]    Loss: 1.074701  Accuracy: 0.63

Test set: Average loss: 1.1052, Accuracy: 6085/10000 (61%)

Train Epoch: 15 [0/50000 (0%)] Loss: 1.043634  Accuracy: 0.62
Train Epoch: 15 [6000/50000 (12%)]     Loss: 1.038746  Accuracy: 0.64
Train Epoch: 15 [12000/50000 (24%)]    Loss: 1.052314  Accuracy: 0.63
Train Epoch: 15 [18000/50000 (36%)]    Loss: 0.971683  Accuracy: 0.63
Train Epoch: 15 [24000/50000 (48%)]    Loss: 1.098648  Accuracy: 0.61
Train Epoch: 15 [30000/50000 (60%)]    Loss: 1.000771  Accuracy: 0.65
Train Epoch: 15 [36000/50000 (71%)]    Loss: 1.055151  Accuracy: 0.64
Train Epoch: 15 [42000/50000 (83%)]    Loss: 1.000841  Accuracy: 0.67
Train Epoch: 15 [48000/50000 (95%)]    Loss: 0.969508  Accuracy: 0.65

Test set: Average loss: 1.0737, Accuracy: 6218/10000 (62%)

Train Epoch: 16 [0/50000 (0%)] Loss: 1.007271  Accuracy: 0.65
Train Epoch: 16 [6000/50000 (12%)]     Loss: 0.960454  Accuracy: 0.65
Train Epoch: 16 [12000/50000 (24%)]    Loss: 1.069016  Accuracy: 0.62
Train Epoch: 16 [18000/50000 (36%)]    Loss: 1.007230  Accuracy: 0.66
Train Epoch: 16 [24000/50000 (48%)]    Loss: 1.030313  Accuracy: 0.64
Train Epoch: 16 [30000/50000 (60%)]    Loss: 1.022426  Accuracy: 0.66
Train Epoch: 16 [36000/50000 (71%)]    Loss: 0.956405  Accuracy: 0.65
Train Epoch: 16 [42000/50000 (83%)]    Loss: 1.018185  Accuracy: 0.67
Train Epoch: 16 [48000/50000 (95%)]    Loss: 1.065085  Accuracy: 0.63

Test set: Average loss: 1.0718, Accuracy: 6190/10000 (62%)

Train Epoch: 17 [0/50000 (0%)] Loss: 0.962336  Accuracy: 0.66
Train Epoch: 17 [6000/50000 (12%)]     Loss: 0.965887  Accuracy: 0.69
Train Epoch: 17 [12000/50000 (24%)]    Loss: 0.954323  Accuracy: 0.69
```

(continues on next page)

(continued from previous page)

```
Train Epoch: 17 [18000/50000 (36%)]    Loss: 1.048035  Accuracy: 0.62
Train Epoch: 17 [24000/50000 (48%)]    Loss: 1.077593  Accuracy: 0.63
Train Epoch: 17 [30000/50000 (60%)]    Loss: 0.927936  Accuracy: 0.68
Train Epoch: 17 [36000/50000 (71%)]    Loss: 0.977208  Accuracy: 0.65
Train Epoch: 17 [42000/50000 (83%)]    Loss: 0.955921  Accuracy: 0.66
Train Epoch: 17 [48000/50000 (95%)]    Loss: 0.982332  Accuracy: 0.66

Test set: Average loss: 1.0708, Accuracy: 6275/10000 (63%)

Train Epoch: 18 [0/50000 (0%)] Loss: 1.109270  Accuracy: 0.62
Train Epoch: 18 [6000/50000 (12%)]     Loss: 0.984331  Accuracy: 0.67
Train Epoch: 18 [12000/50000 (24%)]    Loss: 0.932089  Accuracy: 0.67
Train Epoch: 18 [18000/50000 (36%)]    Loss: 0.974177  Accuracy: 0.65
Train Epoch: 18 [24000/50000 (48%)]    Loss: 0.937838  Accuracy: 0.67
Train Epoch: 18 [30000/50000 (60%)]    Loss: 0.967142  Accuracy: 0.67
Train Epoch: 18 [36000/50000 (71%)]    Loss: 1.013721  Accuracy: 0.63
Train Epoch: 18 [42000/50000 (83%)]    Loss: 0.886761  Accuracy: 0.68
Train Epoch: 18 [48000/50000 (95%)]    Loss: 0.996015  Accuracy: 0.65

Test set: Average loss: 1.0244, Accuracy: 6369/10000 (64%)

Train Epoch: 19 [0/50000 (0%)] Loss: 0.927531  Accuracy: 0.69
Train Epoch: 19 [6000/50000 (12%)]     Loss: 0.959397  Accuracy: 0.67
Train Epoch: 19 [12000/50000 (24%)]    Loss: 0.978619  Accuracy: 0.65
Train Epoch: 19 [18000/50000 (36%)]    Loss: 0.928914  Accuracy: 0.68
Train Epoch: 19 [24000/50000 (48%)]    Loss: 0.926848  Accuracy: 0.66
Train Epoch: 19 [30000/50000 (60%)]    Loss: 0.932224  Accuracy: 0.66
Train Epoch: 19 [36000/50000 (71%)]    Loss: 0.956639  Accuracy: 0.68
Train Epoch: 19 [42000/50000 (83%)]    Loss: 0.944513  Accuracy: 0.67
Train Epoch: 19 [48000/50000 (95%)]    Loss: 1.006125  Accuracy: 0.66

Test set: Average loss: 1.0273, Accuracy: 6382/10000 (64%)
```





Achieving ~75% validation accuracy with our simple CNN Classifier should be possible, how did you do? Let us know!

- Do you see a difference in train accuracy and validation accuracy?
- How does the difference between training and validation accuracy change over time?

The State-of-the-art is currently 99.5% accuracy! (See https://paperswithcode.com/sota/image-classification-on-cifar-10)

One of the areas of improvement is that our model is still not very deep, modern models usually range from 18-50 layers. However, after around ~15 layers, you start to run into some issues with information propagation through your model: the gradient of the loss is not able to reach the first few layers of the model.

Checkout https://arxiv.org/pdf/1512.03385.pdf for the seminal paper fixing this issue.

[1]:
```
import os

from typing import Sequence, Tuple

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from torchvision.utils import make_grid
import numpy as np

%matplotlib inline

DATA_PATH = os.getenv('TEACHER_DIR', os.getcwd()) + '/JHL_data'
```

2.23 Autoencoders

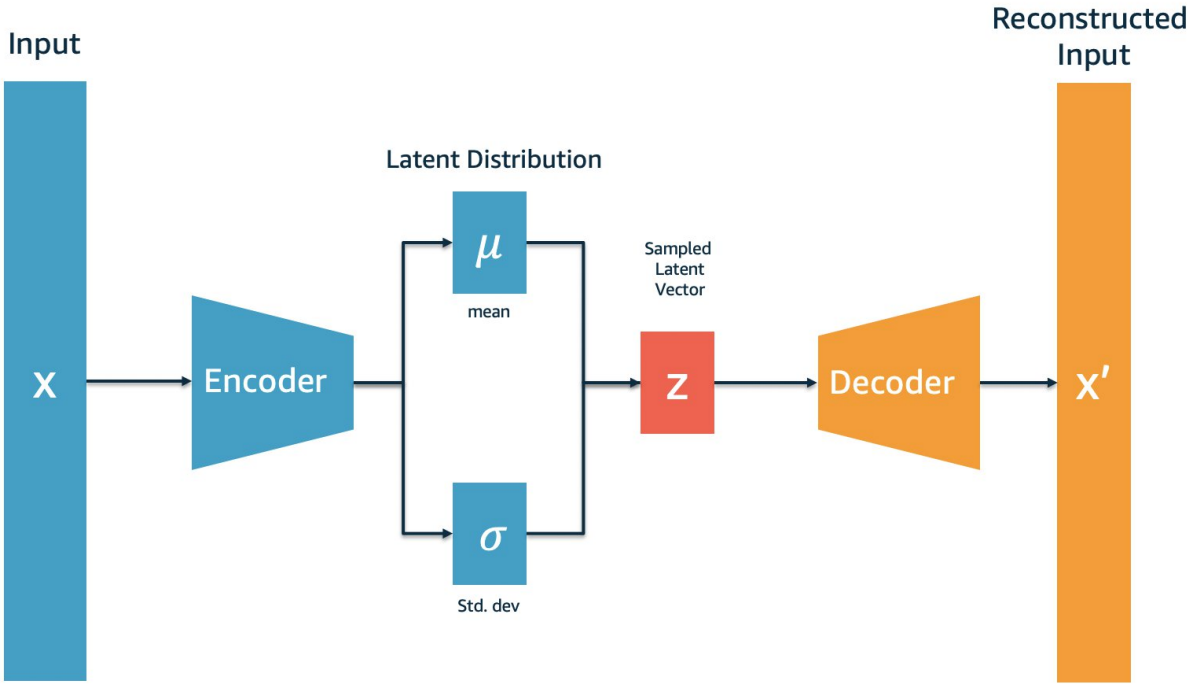
The curse of dimensionality makes it increasingly more difficult to work with highly dimensional data such as images. Increasing the data input dimensions results in an explosion for the computational, storage and data requirements. To achieve reliable results, the amount of data needed grows exponentially with the dimensionality.

In the first hands-on we have seen how a simple feed forward neural network learns to classify digits that live in a 784-dimensional space. Is there a way to describe the same digits but by reducing the input information? Could we somehow convert (map) the 784-dimensional space into a lower dimensional one with minimal information loss?

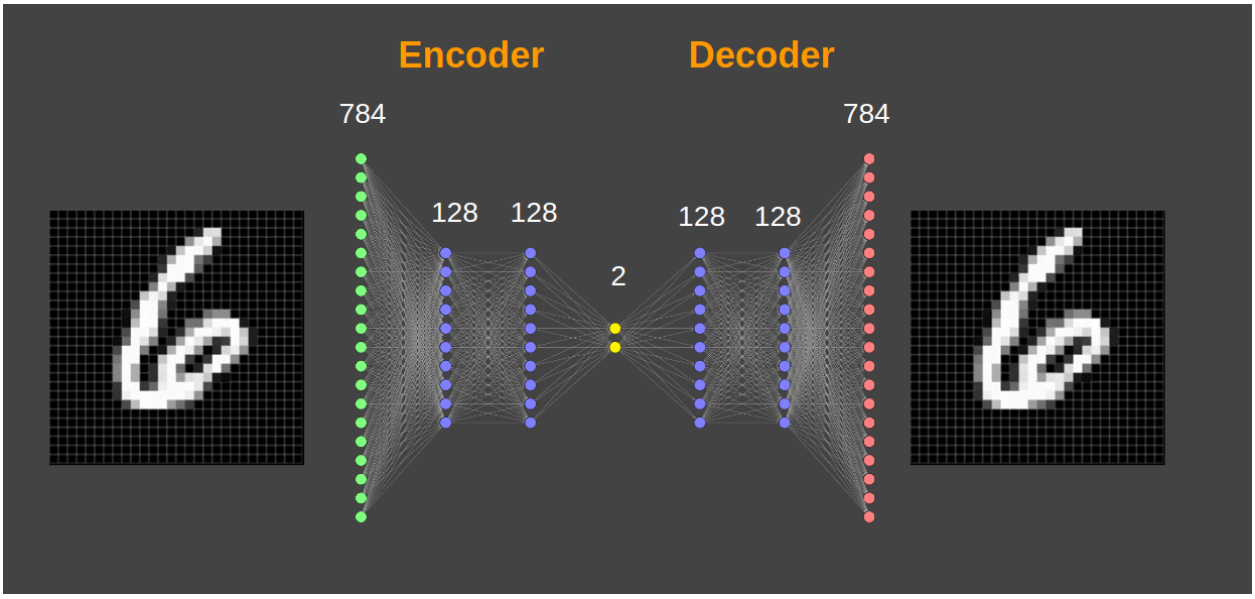
In turns out we can!

2.23.1 The Task for the Autoencoder

With a classifier, the goal is to identify datapoints and label them correctly given the training data.



With an autoencoder we want to *reconstruct* the whole image while first encoding the image into a lower dimensional space.



In the case of the autoencoder, there are no labels or explicit targets. We only have the training images which become both our input and output. The target loss function should then only have to compare the input to the output of the neural network.

Loading Training Data

We can reuse the MNIST data set loading code:

The MNIST dataset is conveniently bundled within Torch Vision, and we can easily take a look at some of its features.

2.23.2 Lets take a look at our input data

[2]:
```
example_trainset = datasets.MNIST(root=DATA_PATH, train=True, transform=transforms.
↪ToTensor(), download=True)

example_image, example_label = next(iter(example_trainset))

print(f"Input data shape: {example_image.shape}")
example_image = example_image[0] # removing first dimension
```

```
Input data shape: torch.Size([1, 28, 28])
```

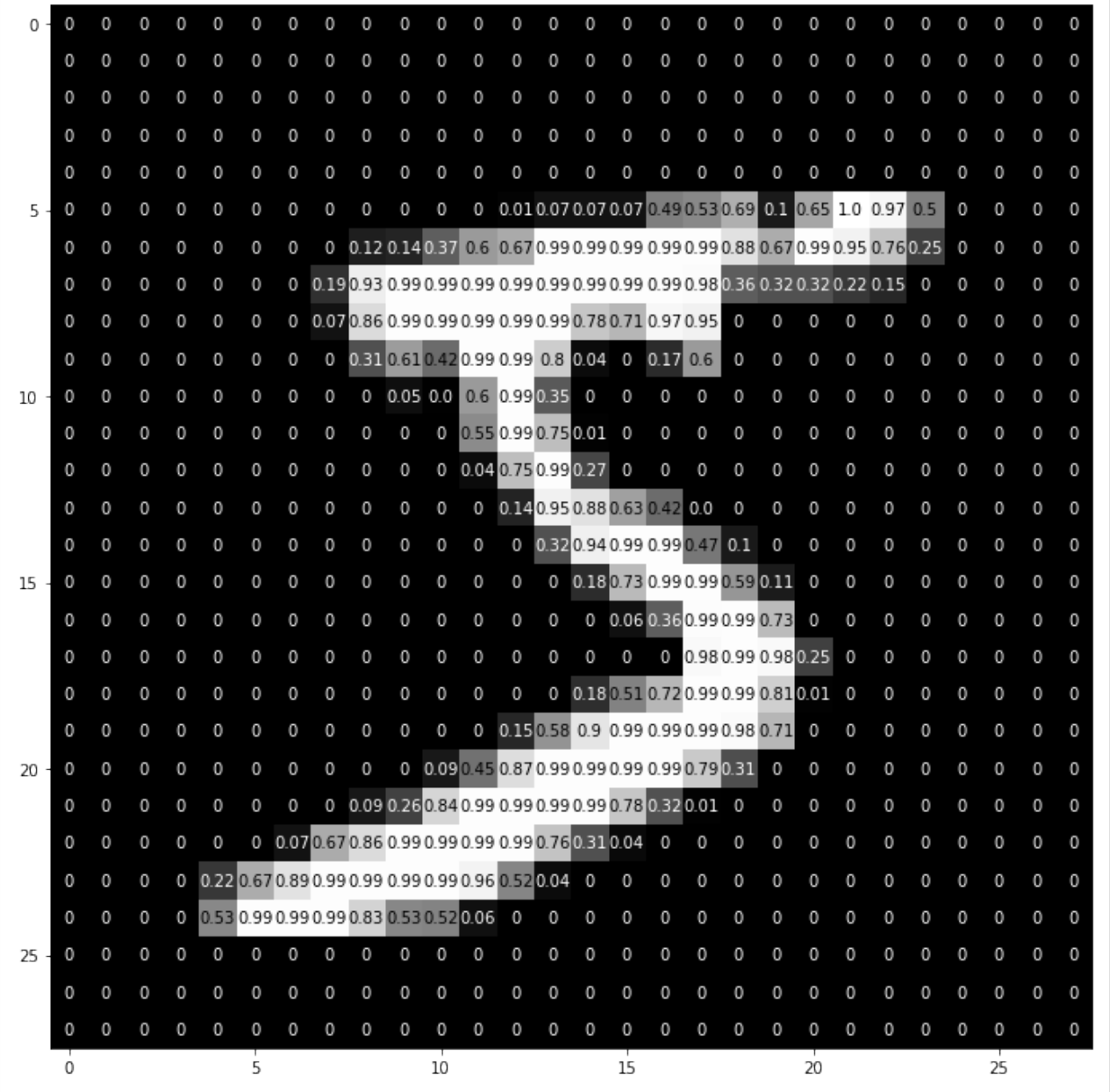
2.23.3 Let's plot the numerical representation of the image so that we can see what the model sees as input!

[3]:
```
fig = plt.figure(figsize = (12,12))
ax = fig.add_subplot(111)
ax.imshow(example_image, cmap='gray')
width, height = example_image.shape
thresh = example_image.max()/2.5
```

(continues on next page)

(continued from previous page)

```
for x in range(width):
    for y in range(height):
        val = round(example_image[x][y].item(), 2) if example_image[x][y] != 0 else 0
        ax.annotate(str(val), xy=(y, x),
                    horizontalalignment='center',
                    verticalalignment='center',
                    color='white' if example_image[x][y]<thresh else 'black')
```



2.24 Formatting the input data layer

Remember that instead of a 28 x 28 matrix, we build our network to accept a 784-length vector.

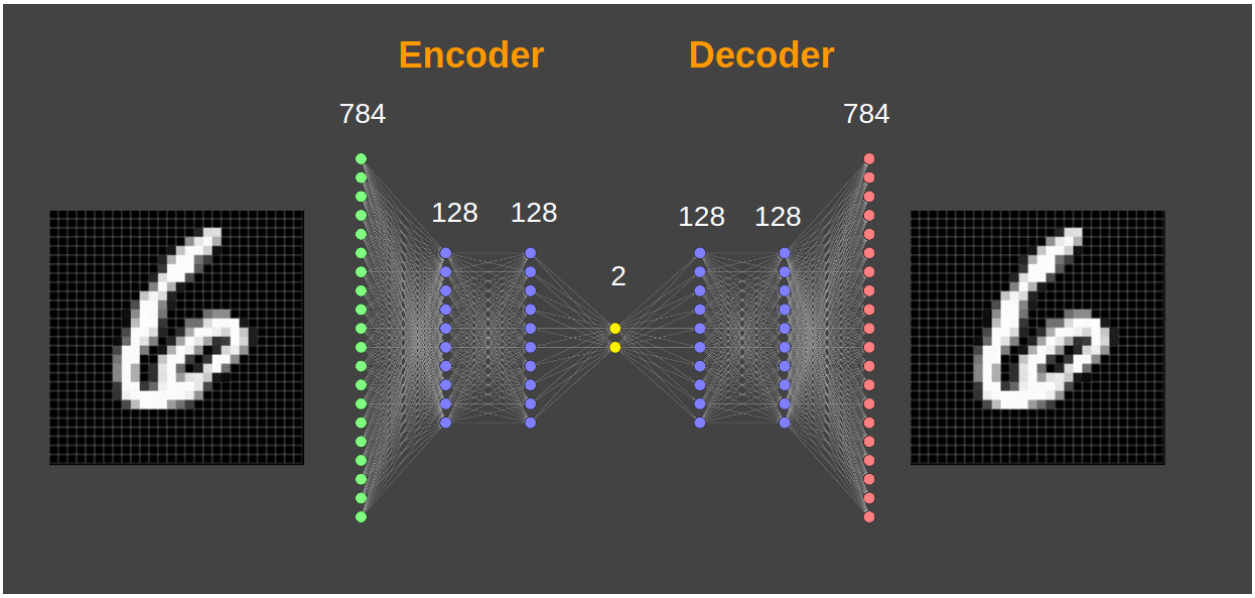
Each image needs to be then reshaped (or flattened) into a vector.

This will still be the first layer to our autoencoder. The autoencoder will essentially learn to reconstruct a vector.

2.25 Building the model

We saw in the lecture how an Autoencoder has a two-part architecture consisting of an Encoder and a Decoder. The encoder is responsible for compressing the input data to a lower space, while the decoder tries to reconstruct the original image from that lower dimensionality.

As such, we will be using the following architecture:



[4]:
```
# We can see our autoencoder as consisting of two neural networks
class Encoder(nn.Module):
    def __init__(self, hidden_dims, latent_dims):
        super(Encoder, self).__init__()
        # input -> flatten -> first later -> second layer -> latent vector
        self.encoder = nn.Sequential(nn.Flatten(), nn.Linear(28*28, hidden_dims), nn.
↪ReLU(),
                                     nn.Linear(hidden_dims,hidden_dims), nn.ReLU())

        self.latent = nn.Linear(hidden_dims, latent_dims)

    def forward(self, x):
        x = self.encoder(x)
        x = self.latent(x)
        return x

# the decoder takes a 2-dimensional vector and reconstructs a digit
```

(continues on next page)

(continued from previous page)

```
class Decoder(nn.Module):
    def __init__(self, latent_dims, hidden_dims):
        super(Decoder, self).__init__()
        # latent vector -> first later -> second layer -> Reconstruction
        self.decoder = nn.Sequential(nn.Linear(latent_dims, hidden_dims), nn.ReLU(),
                                     nn.Linear(hidden_dims,hidden_dims), nn.ReLU())

        # The sigmoid here forces the outputs to be between 0 and 1
        self.output = nn.Sequential(nn.Linear(hidden_dims, 784))

    def forward(self, x):
        x = self.decoder(x)
        x = self.output(x)
        # we reshape our output of 784 to 28x28
        x = x.view(-1,1, 28, 28)
        return x

# the AutoEncoder class 'connects' the encoder and decoder together
class AutoEncoder(nn.Module):
    def __init__(self, hidden_dims, latent_dims):
        super().__init__()  # Python magic which initialises all relevant PyTorch␣
↪properties

        # Here, define your model!
        # We are initializing the encoder and decoder

        # hidden_dims=128, latent=2
        self.encoder = Encoder(hidden_dims, latent_dims)
        self.decoder = Decoder(latent_dims, hidden_dims)


    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

2.25.1 What we want to do to train this model; gradient descent:

Just as the classifier that we trained in the first hands-on; our model is trained using gradient descent. Instead of outputting a probability distribution across ten different digits, we are essentially predicting 784 classes. Each "class" here stands for a pixel value, instead of a class-probability label!

For this purpose we won't be able to use the cross-entropy measure, which is used for categorical outputs (that is, labels). In this task, we want to compare the input with the output. We can simply use the Mean Square error: $\sum(\mathbf{x} - \hat{\mathbf{x}})^2$. Here $\hat{\mathbf{x}}$ is the reconstructed digit and $\mathbf{x}$ is the original digit.

Note how the structure for training this model stays largely the same compared to the first and second hands-on. Remember what we need to train the model:

2.25.2 What we need to train this model under the hood...

- The calculation and retention of computational graphs when you call model.forward()
- The calculation of gradients when we calculate the loss and call loss.backwards()
- The updating of model parameters when we call optimizer.step()

The Test loop

The test loop tells us how well we are reconstructing data points that the model has never seen before! We cannot compute an accuracy, so we print the MSE on the test data and visualize the reconstructions.

[5]:
```
def train(model, device, train_loader, optimizer, epoch, log_interval=10):
    model.train()
    # we don't need the target here!
    for batch_idx, (data, _) in enumerate(train_loader):
        # this is x
        data = data.to(device, non_blocking=True)
        optimizer.zero_grad()

        # this is x hat
        output = model(data)

        plain_autoencoder = not isinstance(output, tuple)
        if plain_autoencoder:
            reconstruction = output
        else:
            reconstruction, kl_div = output

        # MSE loss
        reconstruction_loss = F.mse_loss(reconstruction, data, reduction='sum')

        loss = reconstruction_loss + (0 if plain_autoencoder else kl_div.sum())
        loss.backward()

        optimizer.step()

        if batch_idx % log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()
            ))

        mean_loss = loss.detach().item()/len(data)

        yield ((mean_loss),) if plain_autoencoder else (mean_loss, kl_div.detach().
↪mean().item())

@torch.no_grad()
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    num_samples = 0
```

(continues on next page)

(continued from previous page)

```
    for data, _ in test_loader:
        data = data.to(device, non_blocking=True)

        # get model output
        output = model(data)

        plain_autoencoder = not isinstance(output, tuple)
        if plain_autoencoder:
            reconstruction = output
        else:
            reconstruction, kl_div = output

        # calculate loss
        test_loss += F.mse_loss(reconstruction, data, reduction='sum').item()  # sum up␣
↪batch loss
        num_samples += len(data)

    test_loss /= num_samples
    print('\nTest set: Average loss: {:.4f}'.format(test_loss))

    # visualization, we take the first 16 images
    plt.imshow(
        make_grid(
            torch.cat((data[:9], reconstruction[:9]), dim=0).cpu(),
            normalize=True, nrow=9
        ).permute(1,2,0).numpy()
    )
    plt.show()

    yield ((test_loss,) if plain_autoencoder else (test_loss, kl_div.mean().item()))
```

2.25.3 Function to plot train/validation curves

[6]:
```
def plot_metric_curve(
    train_metric: Sequence[float],
    val_metric: Sequence[float],
    n_epochs: int,
    metric_name: str, # Label of the y-axis, e.g. 'Accuracy'
    x_axis_name: str = 'Epoch'
):
    # create values for the x-axis
    train_steps, val_steps = map(
        lambda metric_values: np.linspace(start=0, stop=n_epochs, num=len(metric_
↪values)),
        (train_metric, val_metric)
    )

    plt.plot(train_steps, train_metric, label='train')
    plt.plot(val_steps, val_metric, label='validation')
    plt.title(f"{metric_name} vs. {x_axis_name}")
```

(continues on next page)

(continued from previous page)

```
    plt.legend()
    plt.show()
```

2.25.4 Function to run training and testing loops

[7]:
```
def fit(model, optimizer, n_epochs, device, train_loader, test_loader, log_interval):

    # get the validation loss and accuracy of the untrained model
    start_val_metrics = tuple(test(model, device, test_loader))[0]

    # don't mind the following train/test loop logic too much, if you want to know what's␣
↪happening, let us know :)
    # normally you would pass a logger to your train/test loops and log the respective␣
↪metrics there
    train_metrics, val_metrics = map(lambda arr: np.asarray(arr).transpose(2,0,1), zip(*[
        (
            [*train(model, device, train_loader, optimizer, epoch, log_interval)],
            [*test(model, device, test_loader)]
        )
        for epoch in range(n_epochs)
    ]))

    # flatten the arrays
    metrics = tuple(map(np.ravel, (*train_metrics, *val_metrics)))

    assert len(metrics) in (2, 4)
    if len(metrics) == 2:  # plain auto-encoder
        train_loss, val_loss = metrics
        val_loss = (start_val_metrics[0], *val_loss)
    elif len(metrics) ==  4: # variational auto-encoder
        train_loss, train_kl, val_loss, val_kl = metrics
        val_loss, val_kl = (start_val_metrics[0], *val_loss), (start_val_metrics[1],␣
↪*val_kl)
        plot_metric_curve(train_kl, val_kl, n_epochs, 'KL-divergence')

    plot_metric_curve(train_loss, val_loss, n_epochs, 'MSE Loss')
```

2.25.5 Training hyperparameters and PyTorch boilerplate

[8]:
```
use_cuda = torch.cuda.is_available()
print(f"CUDA is {'' if use_cuda else 'not '}available")
device = torch.device("cuda" if use_cuda else "cpu")

if use_cuda:
    torch.cuda.set_per_process_memory_fraction(0.22)

cpu_count = len(os.sched_getaffinity(0))
```

```
CUDA is available
```

2.25.6 Training the model

This is the fun part!

The batch size determines over how much data per step is used to compute the loss function, gradients, and back propagation. Large batch sizes allow the network to complete it's training faster; however, there are other factors beyond training speed to consider.

Too large of a batch size reduces the variance of the update step, which may not always be useful for succesfully training the model due to underfitting.

Too small of a batch size increases the variance of the update step, which may lead the optimizer to miss the global minimum.

So a good batch size may take some trial and error to find (or hyperparameter optimization…)!

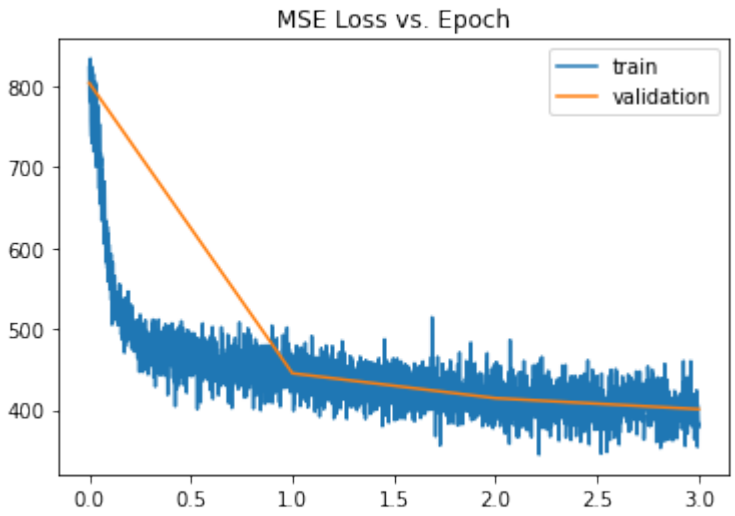
[9]:
```
BATCH_SIZE = 64
EPOCHS = 3
LEARNING_RATE = 1e-4
HIDDEN_DIMS = 128
LATENT_DIMS = 2
LOGGING_INTERVAL = 100

model = AutoEncoder(HIDDEN_DIMS, LATENT_DIMS).to(device)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

transform = transforms.Compose([
    transforms.ToTensor(), # Creates the PyTorch tensors from the PIL images, and␣
↪normalizes them to the [0, 1] interval
    transforms.Normalize((0.1307,), (0.3081,))  # Normalizes the data to 0 mean and 1␣
↪standard deviation
])

train_loader, test_loader = (
    torch.utils.data.DataLoader(
        datasets.MNIST(DATA_PATH, train=train, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=train,
        num_workers=cpu_count
    )
    for train in (True, False)
)

fit(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_INTERVAL)
```

```
Test set: Average loss: 804.0967
```

```
Train Epoch: 0 [0/60000 (0%)]  Loss: 52692.296875
Train Epoch: 0 [6400/60000 (11%)]      Loss: 36425.335938
Train Epoch: 0 [12800/60000 (21%)]     Loss: 31718.781250
Train Epoch: 0 [19200/60000 (32%)]     Loss: 30317.466797
Train Epoch: 0 [25600/60000 (43%)]     Loss: 31041.410156
Train Epoch: 0 [32000/60000 (53%)]     Loss: 30466.101562
Train Epoch: 0 [38400/60000 (64%)]     Loss: 27743.343750
Train Epoch: 0 [44800/60000 (75%)]     Loss: 27713.582031
Train Epoch: 0 [51200/60000 (85%)]     Loss: 28773.500000
Train Epoch: 0 [57600/60000 (96%)]     Loss: 31197.591797

Test set: Average loss: 445.2692
```


```
Train Epoch: 1 [0/60000 (0%)]  Loss: 29973.507812
Train Epoch: 1 [6400/60000 (11%)]      Loss: 26827.787109
Train Epoch: 1 [12800/60000 (21%)]     Loss: 27565.468750
Train Epoch: 1 [19200/60000 (32%)]     Loss: 25871.068359
Train Epoch: 1 [25600/60000 (43%)]     Loss: 27521.066406
Train Epoch: 1 [32000/60000 (53%)]     Loss: 26838.316406
Train Epoch: 1 [38400/60000 (64%)]     Loss: 25902.935547
Train Epoch: 1 [44800/60000 (75%)]     Loss: 26014.958984
Train Epoch: 1 [51200/60000 (85%)]     Loss: 27403.640625
Train Epoch: 1 [57600/60000 (96%)]     Loss: 25713.730469

Test set: Average loss: 414.5186
```


```
Train Epoch: 2 [0/60000 (0%)]  Loss: 27529.267578
Train Epoch: 2 [6400/60000 (11%)]      Loss: 25257.332031
Train Epoch: 2 [12800/60000 (21%)]     Loss: 22058.433594
Train Epoch: 2 [19200/60000 (32%)]     Loss: 24907.625000
Train Epoch: 2 [25600/60000 (43%)]     Loss: 27292.980469
Train Epoch: 2 [32000/60000 (53%)]     Loss: 24695.988281
Train Epoch: 2 [38400/60000 (64%)]     Loss: 25768.566406
Train Epoch: 2 [44800/60000 (75%)]     Loss: 25684.695312
Train Epoch: 2 [51200/60000 (85%)]     Loss: 26965.398438
Train Epoch: 2 [57600/60000 (96%)]     Loss: 29458.785156

Test set: Average loss: 400.9221
```



2.25.7 (Hopefully) your loss goes down, and your reconstructions make sense!

Tinkering with the hyper parameters will let you converge faster (or slower for that matter). After a few iterations the loss will go down and the reconstructions will start to take shape. With the default configuration, it needs a few tens of epochs to start converging. Can you find a better configuration?

Generation using an AutoEncoder.

Now that we have an AutoEncoder well trained, is there a way that we can sample from its (two-dimensional) latent space? This neural network should have learnt how the data looks like when encoded to two dimensions. So, in principle we could randomly generate two real numbers and decode them to 'generate' one digit right? Yes and No.

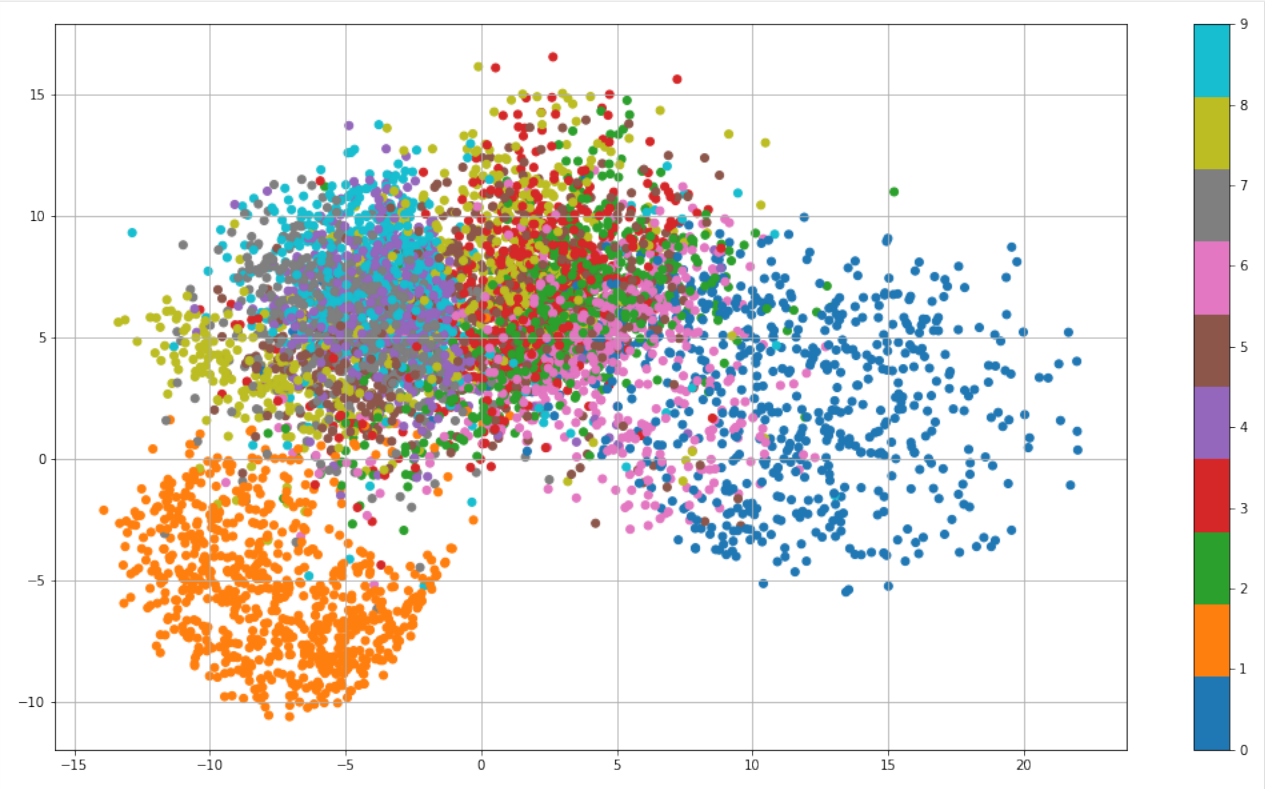
by choosing a point uniformly, by chance, we could find a datapoint that resembles a digit. We never trained the model in a way that guarantees our latent space to be "dense". This means, we will find that our latent space is broken and filled with "gaps" between the datapoints. Furthermore, we are dealing with an easy dataset that is fully representable using a two-dimensional latent space. Not all data can be represented with a low dimensionality and the curse of dimensionality would forbid us to sample something representative in higher dimensions. The space of data in higher dimensions is very sparse.

2.25.8 Inspecting the Latent Space

Since our latent space is just two-dimensional, we can plot this and see how the model has learnt the latent space.

[10]:
```
def plot_latent(model, data, num_batches=100):
    model.eval()
    fig = plt.figure(figsize = (18,10))
    with torch.no_grad():
        for i, (x, y) in enumerate(data):

            # 1. get output from the model
            # 2. if the output contains more elements than just the latents, discard␣
↪these
            # 3. convert pytorch tensor to numpy arrays
            z_latent = (
                out[0]
                if isinstance(
                    (out := model.encoder(x.to(device))),
                    tuple
                )
                else out
            ).cpu().numpy()

            plt.scatter(z_latent[:, 0], z_latent[:, 1], c=y, cmap='tab10')
            if i > num_batches:
                plt.colorbar()
                plt.grid()
                break
```

[11]:
```
plot_latent(model, test_loader)
```



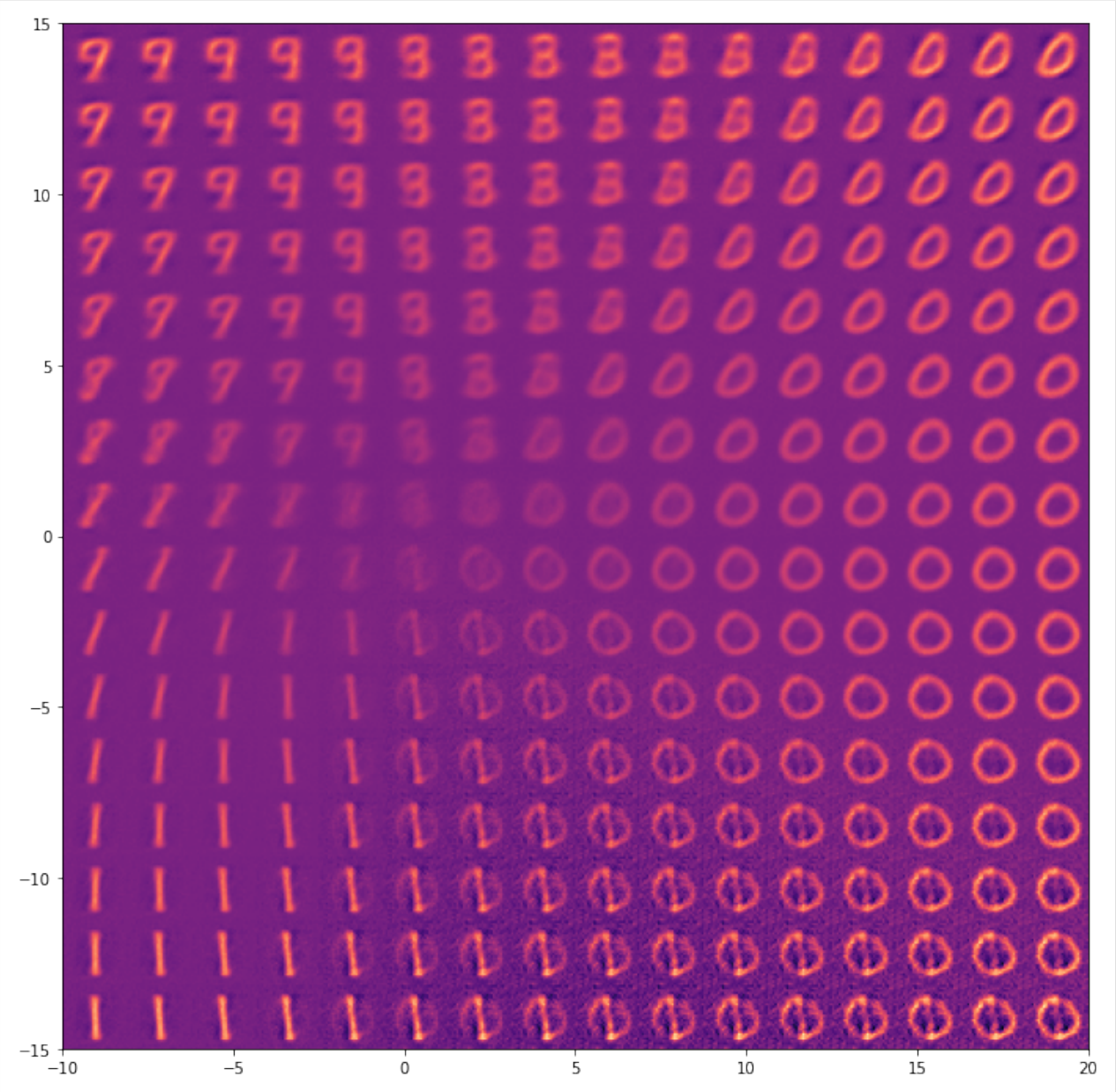
The Learned Latent Space

We can see that there is a lot of overlap between the digit classes. For instance, we can see here the model has difficulty segregating the digit 9 (light blue) from the digit 4 (purple). Ideally, we want these classes to be separated (disentangled) as much as possible from each other.

We can attempt to 'walk' or traverse this learned latent space by uniformly sampling points in sequence:

[12]:
```
def plot_reconstructed(model, r0=(0, 20), r1=(-5, 15), n=16):
    model.eval()
    fig = plt.figure(figsize = (12,12))
    w = 28
    img = np.zeros((n*w, n*w))
    with torch.no_grad():
        for i, y in enumerate(np.linspace(*r1, n)):
            for j, x in enumerate(np.linspace(*r0, n)):
                latent = torch.Tensor([[x, y]]).to(device)
                x_hat = model.decoder(latent)
                x_hat = x_hat.reshape(28, 28).cpu().numpy()
                img[(n-1-i)*w:(n-1-i+1)*w, j*w:(j+1)*w] = x_hat
        plt.imshow(img, extent=[*r0, *r1], cmap='magma')
```

[13]:
```
plot_reconstructed(model, r0=(-10, 20), r1=(-15, 15))
```

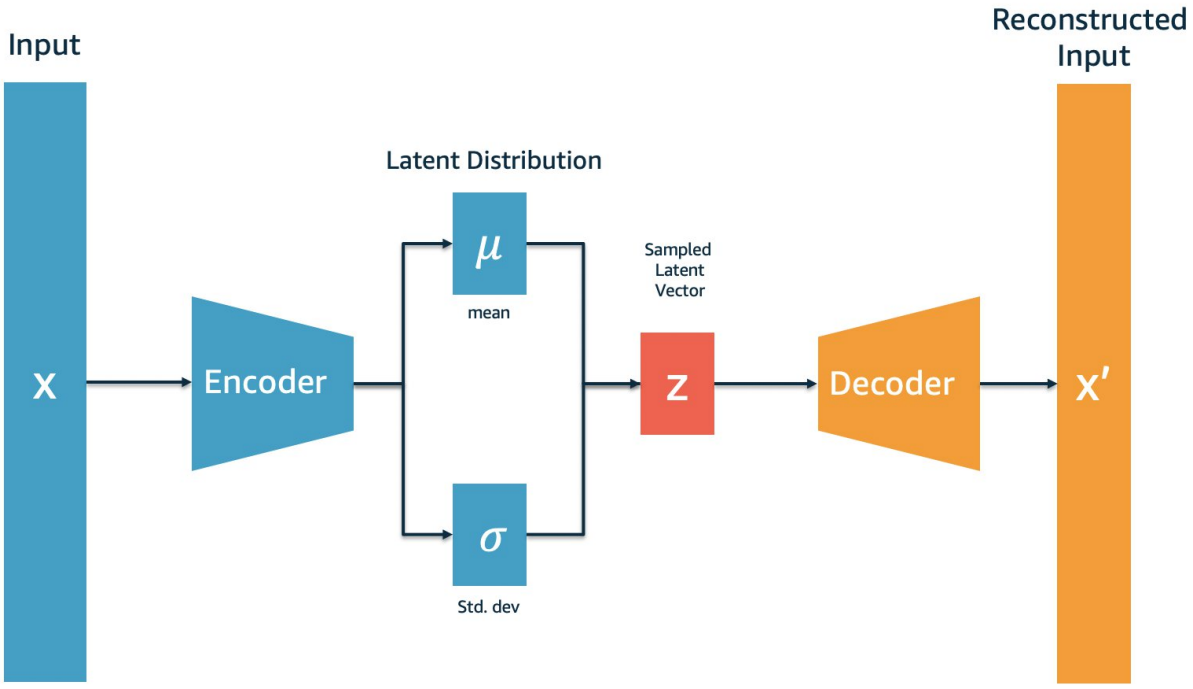


Here we see roughly how this traversal takes the shape as the figure above of the learned latent space. Notice how the shapes of the digits between 9, 4 and 0 are incongruous. In the plot we see that there are holes in the latent space. For instance, the model does not "know" what the relationship is between 7 and 1. We need something more powerful to learn the data distribution better and to somehow teach the model to learn a smoother data distribution.

2.26 Variational Auto Encoder

As we saw during the lecture. The (vanilla) VAE looks like a regular regular autoencoder except that we try to learn the data statistics. We do this by parameterizing the Gaussian (Normal) distribution. That is, while an autoencoder learns a deterministic latent, the VAE learns the means and standard deviations of the latent space; a probability distribution.

The output of the encoder part is replaced by a mean output $\mu(\mathbf{x})$ and a standard deviation output $\sigma(\mathbf{x})$. The decoder takes as input a latent vector sampled from a normal distribution using the learned $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$. In essence, we do not need to change the decoder class that we already have, only the encoder.



[14]:
```
# We can largely reuse our previous Encoder, while the decoder remains unchanged.
class VariationalEncoder(nn.Module):
    def __init__(self, hidden_dims, latent_dims):
        super(VariationalEncoder, self).__init__()
        # input -> flatten -> first later -> second layer -> (means, stds)
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, hidden_dims),
            nn.ReLU(),
            nn.Linear(hidden_dims, 2*latent_dims)
        )


    def forward(self, x):
        mean, logvar = torch.chunk(self.encoder(x), chunks=2, dim=1)
```

(continues on next page)

(continued from previous page)

```
        # reparemeterization trick
        z_latent = mean + (0.5 * logvar).exp() * torch.randn_like(mean)

        return z_latent, mean, logvar

class VAE(nn.Module):
    def __init__(self, hidden_dims, latent_dims):
        super(VAE, self).__init__()
        self.latent_dims = latent_dims
        self.encoder = VariationalEncoder(hidden_dims, latent_dims)
        self.decoder = Decoder(latent_dims, hidden_dims)

    def forward(self, x):
        z_latent, mu, logvar = self.encoder(x)
        kl_loss = -0.5 * torch.sum(1 + logvar - mu ** 2 - logvar.exp(), dim=-1)
        decoded = self.decoder(z_latent if self.training else mu)
        return decoded, kl_loss

    def generate(self, n_digits=16):

        sample = torch.distributions.Normal(0, 1).sample((n_digits,self.latent_dims))
        generated = self.decoder(sample.to(next(iter(self.parameters())).device))
        return generated
```

2.27 Training a VAE

Now that we have defined our VAE, we can train this as we saw in the slides. We can reuse the code that above that we used for the autoencoder. However, there are a few changes we have to make such as adding the KL divergence penalty term to force our network to keep close to the standard normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{1})$.

Remember that this penalty term is needed for encouraging the model to learn a latent space to be more like the standard normal distribution. In this manner we are also disentangling the space more. In our Autoencoder we had the problem that some digit classes overlapped a lot. With a KLD penalty term we are forcing the model to learn the meaningful (separating) properties that constitute the digits. Keep in mind that both the KLD penalty term and the reconstruction loss (MSE) are needed for a smooth representation of the latent space.

[15]:
```
BATCH_SIZE = 128
EPOCHS = 25
LEARNING_RATE = 1e-3
LOG_INTERVAL = 600
HIDDEN_DIMS = 128
LATENT_DIMS = 2

model = VAE(HIDDEN_DIMS, LATENT_DIMS).to(device)
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)

transform = transforms.Compose([
    transforms.ToTensor(), # Creates the PyTorch tensors from the PIL images, and␣
↪normalizes them to the [0, 1] interval
    transforms.Normalize((0.1307,), (0.3081,))  # Normalizes the data to 0 mean and 1␣
↪standard deviation
```

(continues on next page)

(continued from previous page)

```
])

train_loader, test_loader = (
    torch.utils.data.DataLoader(
        datasets.MNIST(DATA_PATH, train=train, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=True,
        shuffle=train
    )
    for train in (True, False)
)

fit(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_INTERVAL)
```

```
Test set: Average loss: 803.3622
```


```
Train Epoch: 0 [0/60000 (0%)]  Loss: 100331.460938
Train Epoch: 0 [12800/60000 (21%)]     Loss: 58773.906250
Train Epoch: 0 [25600/60000 (43%)]     Loss: 57628.613281
Train Epoch: 0 [38400/60000 (64%)]     Loss: 53894.015625
Train Epoch: 0 [51200/60000 (85%)]     Loss: 51857.000000

Test set: Average loss: 396.2138
```


```
Train Epoch: 1 [0/60000 (0%)]  Loss: 50635.582031
Train Epoch: 1 [12800/60000 (21%)]     Loss: 51178.019531
Train Epoch: 1 [25600/60000 (43%)]     Loss: 48155.000000
Train Epoch: 1 [38400/60000 (64%)]     Loss: 49482.285156
Train Epoch: 1 [51200/60000 (85%)]     Loss: 46709.148438

Test set: Average loss: 365.2770
```


```
Train Epoch: 2 [0/60000 (0%)]  Loss: 50180.742188
Train Epoch: 2 [12800/60000 (21%)]     Loss: 47753.808594
Train Epoch: 2 [25600/60000 (43%)]     Loss: 45166.710938
Train Epoch: 2 [38400/60000 (64%)]     Loss: 45727.886719
Train Epoch: 2 [51200/60000 (85%)]     Loss: 47092.417969

Test set: Average loss: 354.1399
```


```
Train Epoch: 3 [0/60000 (0%)]  Loss: 48886.679688
Train Epoch: 3 [12800/60000 (21%)]     Loss: 45953.519531
Train Epoch: 3 [25600/60000 (43%)]     Loss: 46031.453125
Train Epoch: 3 [38400/60000 (64%)]     Loss: 46320.082031
Train Epoch: 3 [51200/60000 (85%)]     Loss: 44965.007812

Test set: Average loss: 345.9133
```


```
Train Epoch: 4 [0/60000 (0%)]  Loss: 48194.992188
Train Epoch: 4 [12800/60000 (21%)]     Loss: 44277.726562
Train Epoch: 4 [25600/60000 (43%)]     Loss: 42656.433594
Train Epoch: 4 [38400/60000 (64%)]     Loss: 45718.191406
Train Epoch: 4 [51200/60000 (85%)]     Loss: 43374.664062

Test set: Average loss: 338.0396
```

```
Train Epoch: 5 [0/60000 (0%)]  Loss: 41751.433594
Train Epoch: 5 [12800/60000 (21%)]     Loss: 45438.218750
Train Epoch: 5 [25600/60000 (43%)]     Loss: 45285.824219
Train Epoch: 5 [38400/60000 (64%)]     Loss: 44841.144531
Train Epoch: 5 [51200/60000 (85%)]     Loss: 44653.878906

Test set: Average loss: 335.3121
```


```
Train Epoch: 6 [0/60000 (0%)]  Loss: 42858.382812
Train Epoch: 6 [12800/60000 (21%)]     Loss: 44202.128906
Train Epoch: 6 [25600/60000 (43%)]     Loss: 42199.125000
Train Epoch: 6 [38400/60000 (64%)]     Loss: 45075.031250
Train Epoch: 6 [51200/60000 (85%)]     Loss: 41245.703125

Test set: Average loss: 331.5404
```


```
Train Epoch: 7 [0/60000 (0%)]  Loss: 42468.445312
Train Epoch: 7 [12800/60000 (21%)]     Loss: 41186.593750
Train Epoch: 7 [25600/60000 (43%)]     Loss: 46754.531250
Train Epoch: 7 [38400/60000 (64%)]     Loss: 44102.222656
Train Epoch: 7 [51200/60000 (85%)]     Loss: 44692.625000

Test set: Average loss: 329.8199
```


```
Train Epoch: 8 [0/60000 (0%)]  Loss: 44126.085938
Train Epoch: 8 [12800/60000 (21%)]     Loss: 44146.699219
Train Epoch: 8 [25600/60000 (43%)]     Loss: 43232.527344
Train Epoch: 8 [38400/60000 (64%)]     Loss: 44369.648438
Train Epoch: 8 [51200/60000 (85%)]     Loss: 42283.285156

Test set: Average loss: 326.8928
```


```
Train Epoch: 9 [0/60000 (0%)]  Loss: 42653.445312
Train Epoch: 9 [12800/60000 (21%)]     Loss: 42459.523438
Train Epoch: 9 [25600/60000 (43%)]     Loss: 41533.667969
Train Epoch: 9 [38400/60000 (64%)]     Loss: 43704.007812
Train Epoch: 9 [51200/60000 (85%)]     Loss: 39370.125000

Test set: Average loss: 324.0493
```


```
Train Epoch: 10 [0/60000 (0%)] Loss: 41907.304688
Train Epoch: 10 [12800/60000 (21%)]    Loss: 41627.371094
Train Epoch: 10 [25600/60000 (43%)]    Loss: 42345.160156
Train Epoch: 10 [38400/60000 (64%)]    Loss: 43164.593750
Train Epoch: 10 [51200/60000 (85%)]    Loss: 40361.593750

Test set: Average loss: 322.5701
```


```
Train Epoch: 11 [0/60000 (0%)] Loss: 42707.953125
Train Epoch: 11 [12800/60000 (21%)]    Loss: 42216.078125
Train Epoch: 11 [25600/60000 (43%)]    Loss: 40804.328125
Train Epoch: 11 [38400/60000 (64%)]    Loss: 41371.644531
Train Epoch: 11 [51200/60000 (85%)]    Loss: 41022.835938

Test set: Average loss: 319.8355
```

```
Train Epoch: 12 [0/60000 (0%)] Loss: 38908.261719
Train Epoch: 12 [12800/60000 (21%)]    Loss: 41509.039062
Train Epoch: 12 [25600/60000 (43%)]    Loss: 42643.828125
Train Epoch: 12 [38400/60000 (64%)]    Loss: 43944.187500
Train Epoch: 12 [51200/60000 (85%)]    Loss: 41541.625000

Test set: Average loss: 320.3316
```


```
Train Epoch: 13 [0/60000 (0%)] Loss: 42304.234375
Train Epoch: 13 [12800/60000 (21%)]    Loss: 45502.914062
Train Epoch: 13 [25600/60000 (43%)]    Loss: 40167.167969
Train Epoch: 13 [38400/60000 (64%)]    Loss: 41807.335938
Train Epoch: 13 [51200/60000 (85%)]    Loss: 42482.242188

Test set: Average loss: 317.1793
```


```
Train Epoch: 14 [0/60000 (0%)] Loss: 39240.882812
Train Epoch: 14 [12800/60000 (21%)]    Loss: 43356.062500
Train Epoch: 14 [25600/60000 (43%)]    Loss: 40428.488281
Train Epoch: 14 [38400/60000 (64%)]    Loss: 41335.316406
Train Epoch: 14 [51200/60000 (85%)]    Loss: 41421.984375

Test set: Average loss: 317.6060
```


```
Train Epoch: 15 [0/60000 (0%)] Loss: 40953.937500
Train Epoch: 15 [12800/60000 (21%)]    Loss: 42403.468750
Train Epoch: 15 [25600/60000 (43%)]    Loss: 41228.554688
Train Epoch: 15 [38400/60000 (64%)]    Loss: 39591.375000
Train Epoch: 15 [51200/60000 (85%)]    Loss: 42836.894531

Test set: Average loss: 316.0814
```

```
Train Epoch: 16 [0/60000 (0%)] Loss: 40018.421875
Train Epoch: 16 [12800/60000 (21%)]    Loss: 43165.976562
Train Epoch: 16 [25600/60000 (43%)]    Loss: 42580.253906
Train Epoch: 16 [38400/60000 (64%)]    Loss: 42101.179688
Train Epoch: 16 [51200/60000 (85%)]    Loss: 42016.296875

Test set: Average loss: 314.2656
```

```
Train Epoch: 17 [0/60000 (0%)] Loss: 36748.492188
Train Epoch: 17 [12800/60000 (21%)]    Loss: 42645.652344
Train Epoch: 17 [25600/60000 (43%)]    Loss: 41518.980469
Train Epoch: 17 [38400/60000 (64%)]    Loss: 38376.613281
Train Epoch: 17 [51200/60000 (85%)]    Loss: 41745.730469

Test set: Average loss: 314.0731
```

```
Train Epoch: 18 [0/60000 (0%)] Loss: 40707.019531
Train Epoch: 18 [12800/60000 (21%)]    Loss: 43637.886719
Train Epoch: 18 [25600/60000 (43%)]    Loss: 39138.968750
Train Epoch: 18 [38400/60000 (64%)]    Loss: 41260.230469
Train Epoch: 18 [51200/60000 (85%)]    Loss: 38477.539062

Test set: Average loss: 314.2041
```

```
Train Epoch: 19 [0/60000 (0%)] Loss: 38245.265625
Train Epoch: 19 [12800/60000 (21%)]    Loss: 40816.242188
Train Epoch: 19 [25600/60000 (43%)]    Loss: 38803.304688
Train Epoch: 19 [38400/60000 (64%)]    Loss: 40540.101562
Train Epoch: 19 [51200/60000 (85%)]    Loss: 42938.121094

Test set: Average loss: 312.3818
```

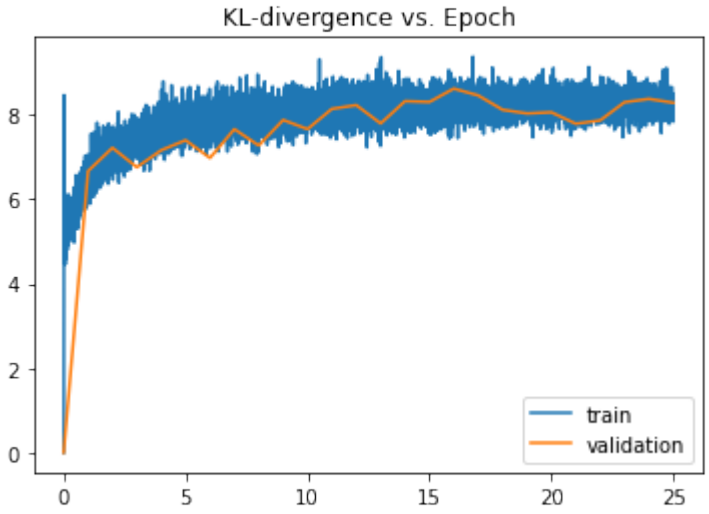

```
Train Epoch: 20 [0/60000 (0%)] Loss: 41309.234375
Train Epoch: 20 [12800/60000 (21%)]    Loss: 40849.363281
Train Epoch: 20 [25600/60000 (43%)]    Loss: 42497.843750
Train Epoch: 20 [38400/60000 (64%)]    Loss: 41567.335938
Train Epoch: 20 [51200/60000 (85%)]    Loss: 37434.992188

Test set: Average loss: 311.6655
```

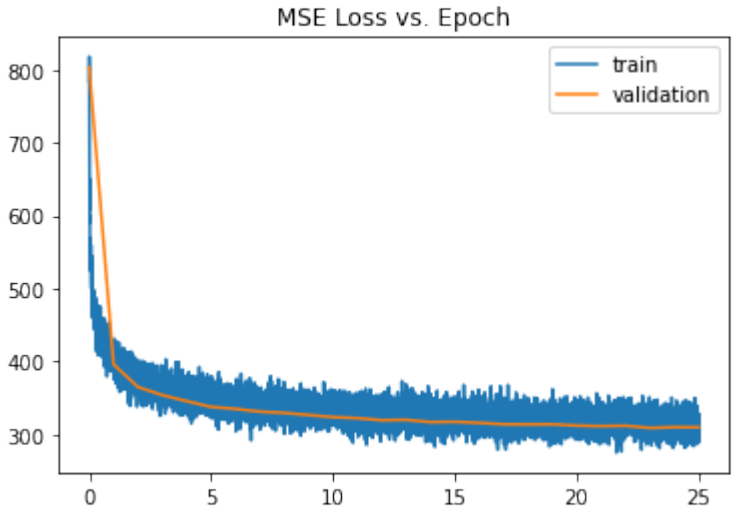

```
Train Epoch: 21 [0/60000 (0%)] Loss: 40542.035156
Train Epoch: 21 [12800/60000 (21%)]    Loss: 38623.675781
Train Epoch: 21 [25600/60000 (43%)]    Loss: 39974.218750
Train Epoch: 21 [38400/60000 (64%)]    Loss: 39486.484375
Train Epoch: 21 [51200/60000 (85%)]    Loss: 38037.890625

Test set: Average loss: 312.2146
```


```
Train Epoch: 22 [0/60000 (0%)] Loss: 41980.058594
Train Epoch: 22 [12800/60000 (21%)]    Loss: 39841.605469
Train Epoch: 22 [25600/60000 (43%)]    Loss: 40483.671875
Train Epoch: 22 [38400/60000 (64%)]    Loss: 40247.746094
Train Epoch: 22 [51200/60000 (85%)]    Loss: 40162.238281

Test set: Average loss: 309.1719
```


```
Train Epoch: 23 [0/60000 (0%)] Loss: 42240.218750
Train Epoch: 23 [12800/60000 (21%)]    Loss: 40843.945312
Train Epoch: 23 [25600/60000 (43%)]    Loss: 42905.046875
Train Epoch: 23 [38400/60000 (64%)]    Loss: 41231.726562
Train Epoch: 23 [51200/60000 (85%)]    Loss: 41236.085938

Test set: Average loss: 310.2775
```


```
Train Epoch: 24 [0/60000 (0%)] Loss: 39683.523438
Train Epoch: 24 [12800/60000 (21%)]    Loss: 42409.585938
Train Epoch: 24 [25600/60000 (43%)]    Loss: 40022.445312
Train Epoch: 24 [38400/60000 (64%)]    Loss: 41567.242188
Train Epoch: 24 [51200/60000 (85%)]    Loss: 39735.773438

Test set: Average loss: 310.1293
```

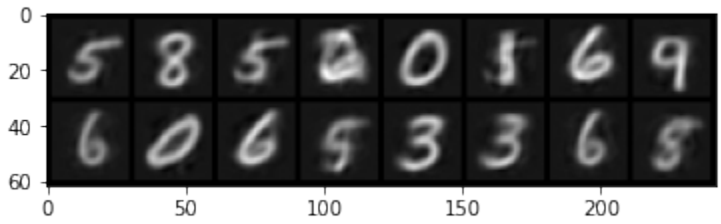




2.28 Generation!

We can now generate from the latent space by sampling from the unit normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{1})$. We first sample a two-dimensional latent and then decode it to obtain digits.

[16]:
```
model.eval()
with torch.no_grad():
    generated_digits = model.generate(16)
    # visualization
    generated_digits = generated_digits.cpu()
    generated_digits_grid = make_grid(generated_digits, normalize=True).permute(1,2,0).
↪numpy()

    plt.imshow(generated_digits_grid)
    plt.show()
```

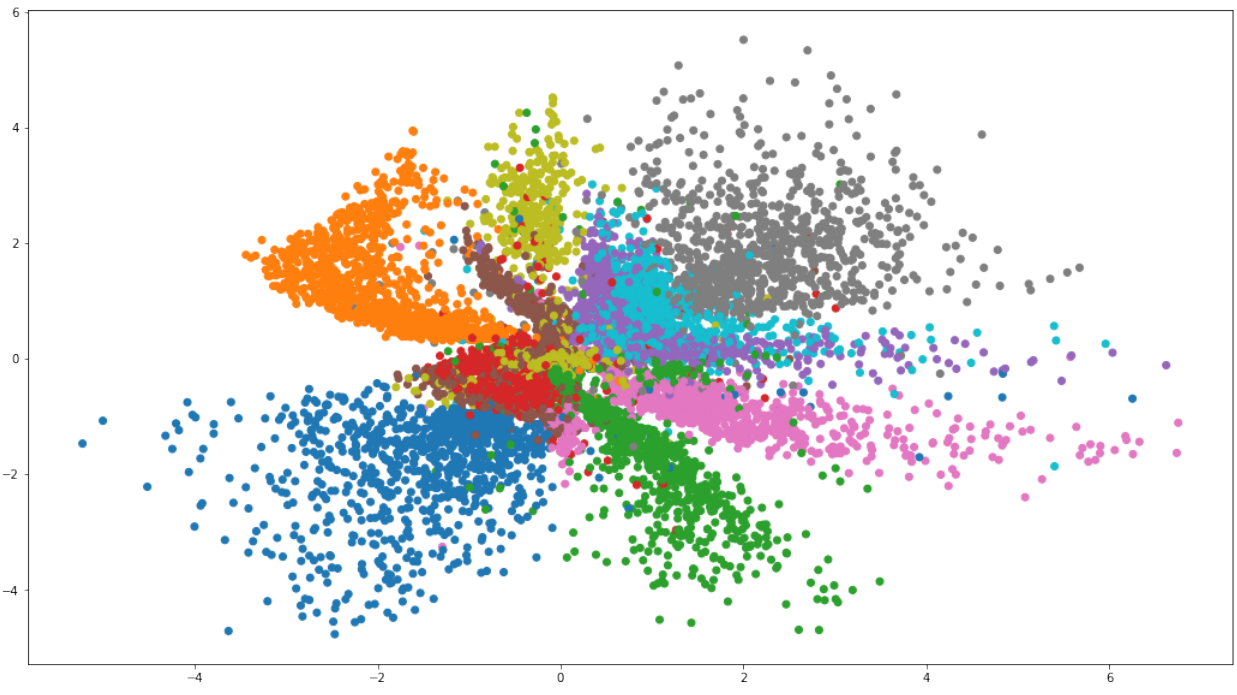


2.29 Let's inspect the VAE latent space

By running the cell below we can peek into the latent space that has been learned by the model. In an instant we see what the addition of the KLD term did to our latent space:

1. The latent space is more disentangled; we can see how (almost) each class has been separated from the rest, and we see this in the generations as well; we can generate better looking data points than we did with the autoencoder.
2. The shape of the latent space seems to be centered and more uniform than the latent space of the autoencoder. This is effectively the purpose of the KLD term; we forced the neural network to approximate the data distribution statistics with a unit gaussian!

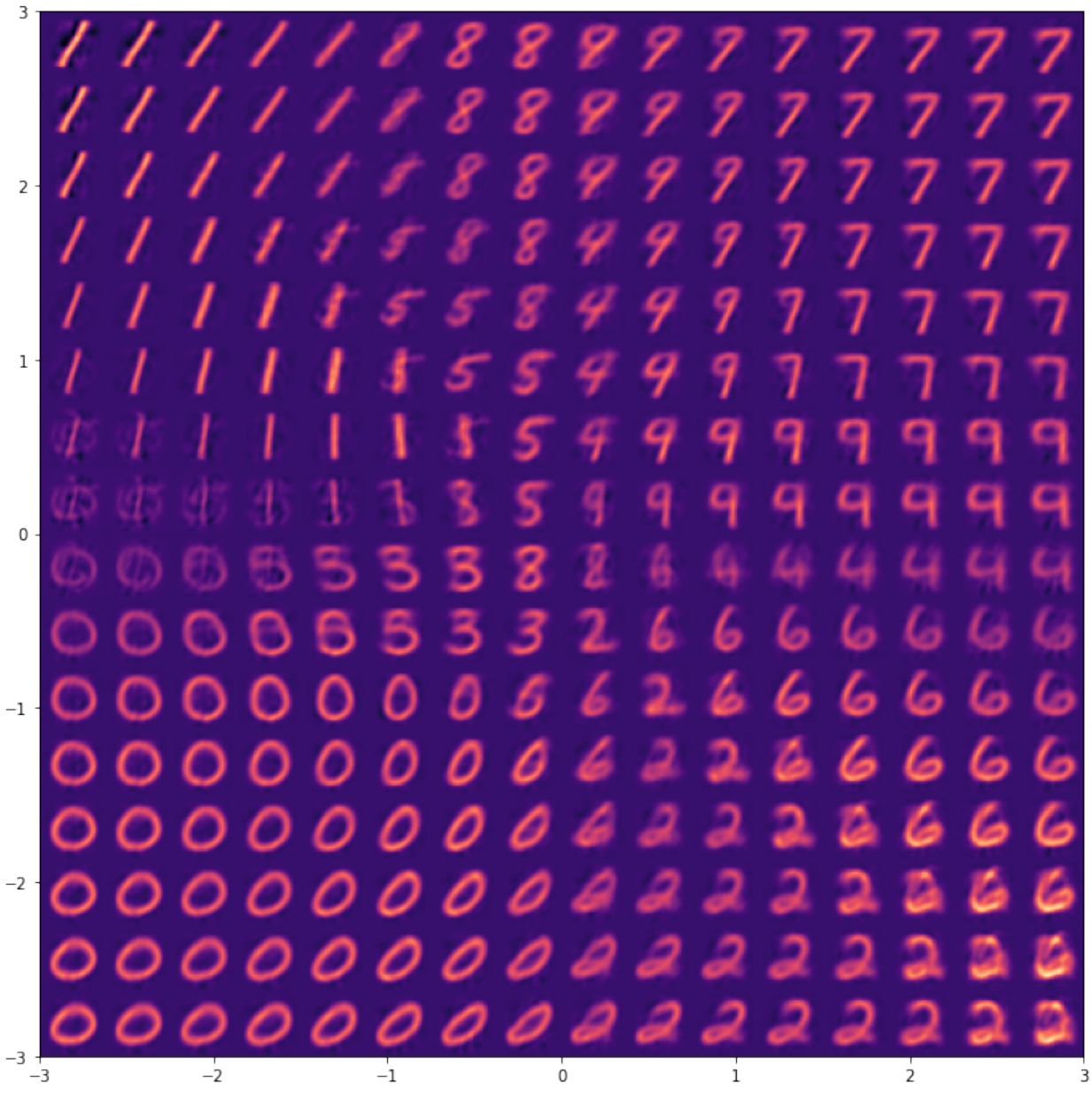
[17]:
```
plot_latent(model, test_loader)
```



2.30 Smooth transitions in our latent space

Let's plot the traversal from one digit to another! We should see no gaps and a smoother transition between the digits!

[18]:
```
plot_reconstructed(model, r0=(-3, 3), r1=(-3, 3))
```



2.31 Conclusion

We can reconstruct data using neural networks but the reconstruction does not imply generative capabilities. We have seen how we should engineer an architecture to add generative capabilities by using variational inference. We have seen how a VAE presents us useful properties that help in distinguishing features that identify certain classes without explicitly training on those classes. VAEs simply offer a more disentangled and smooth latent space from which we can generate data points easily.

Now you should try it out! Change whatever parameter, add or remove layers and see what happens. Perhaps try convolutional layers? VAEs are easy to train but you still have to fine tune properly to get some clean results.

2.32 EXTRA: 3D Projection Plotting

We have essentially compressed our data into two dimensions. We lose a lot of information by doing this hard encoding. We can choose to train with bigger latent sizes, but we would lose the ability to plot these and inspect the latent space visually!

If we retrain our model with a latent_dim of 3,visualizing it using a planar plot would mean discarding the information of the third dimension. However, by adding an axis for the third dimension we can visualize the space nevertheless!

[19]:
```
def plot_3d_latent(model, data, num_batches=100):
    model.eval()
    fig = plt.figure(figsize = (18,10))
    ax = fig.add_subplot(projection='3d')
    with torch.no_grad():
        for i, (x, y) in enumerate(data):
            # extract the latent

            z_latent, _, _ = model.encoder(x.to(device))

            # visualize the 3D latent
            z_latent = z_latent.to(device).cpu().numpy()
            ax.scatter(z_latent[:, 0], z_latent[:, 1], z_latent[:,2], c=y, cmap='tab10')
            if i > num_batches:
                plt.grid()
                break
        ax.set_xlabel('X Label')
        ax.set_ylabel('Y Label')
        ax.set_zlabel('Z Label')
```

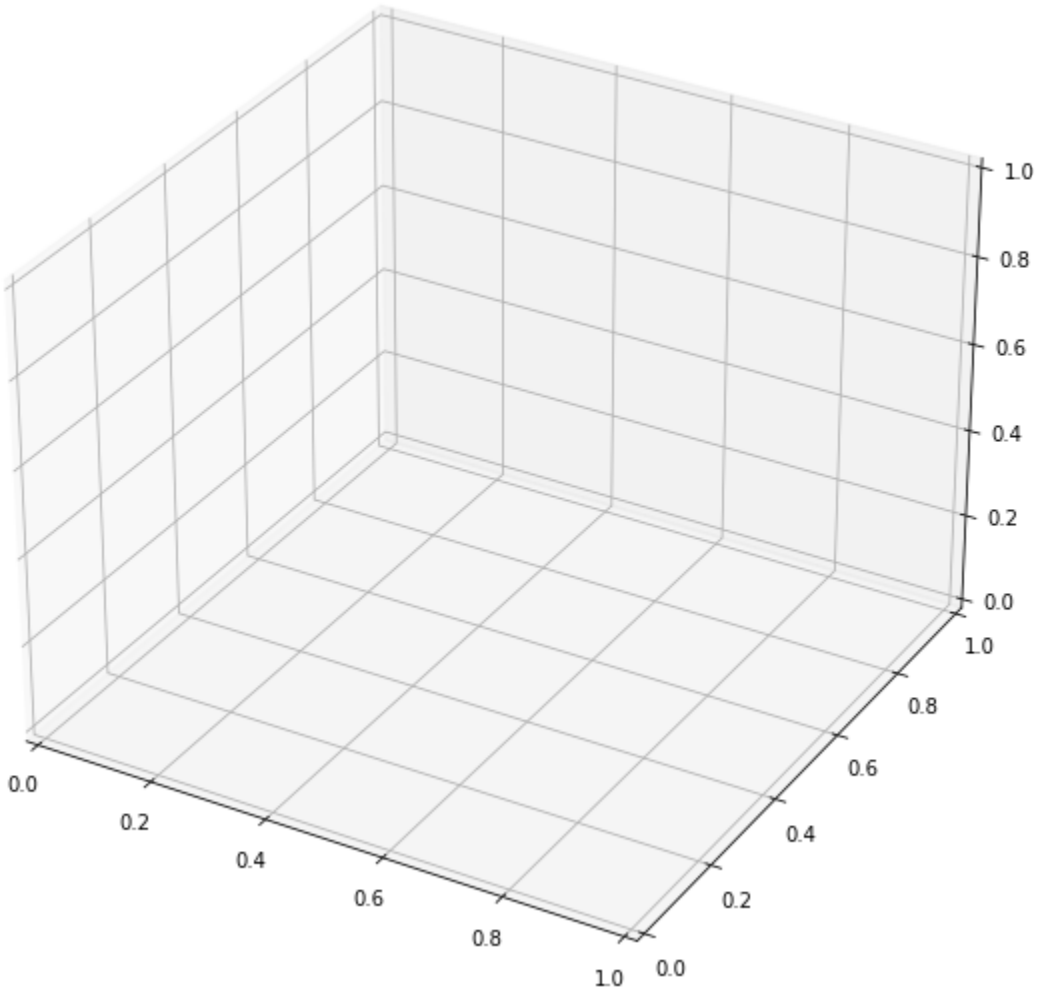
[20]:
```
plot_3d_latent(model, test_loader)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_30612/3682840096.py in <module>
----> 1 plot_3d_latent(model, test_loader)

/tmp/ipykernel_30612/793059455.py in plot_3d_latent(model, data, num_batches)
     11             # visualize the 3D latent
     12             z_latent = z_latent.to(device).cpu().numpy()
---> 13             ax.scatter(z_latent[:, 0], z_latent[:, 1], z_latent[:,2], c=y, cmap=
↪'tab10')
```

(continues on next page)

(continued from previous page)

```
     14             if i > num_batches:
     15                 plt.grid()

IndexError: index 2 is out of bounds for axis 1 with size 2
```



2.33 References and more Learning!

https://arxiv.org/pdf/1312.6114.pdf

1. https://jaan.io/what-is-variational-autoencoder-vae-tutorial/
2. https://avandekleut.github.io/vae/
3. https://hal.archives-ouvertes.fr/hal-02266937/file/LXAI%20at%20ICML%20-%20Oral%20Presentation.pdf
4. https://www.geeksforgeeks.org/implementing-an-autoencoder-in-pytorch/

5. https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial9/AE_CIFAR10.html