Lab 0 - Explainable and Trustworthy AI

Teaching Assistant: *Gabriele Ciravegna*

Lab 0 (2): Introduction to Deep Learning in Pytorch

PyTorch provides the elegantly designed modules and classes torch.nn, torch.optim, Dataset, and DataLoader to help you create and train neural networks.

In order to fully utilize their power and customize them for your problem, you need to really understand exactly what they're doing. To develop this understanding, we will first train basic neural net on the digits data set without using any features from these models; we will initially only use the most basic PyTorch tensor functionality. Then, we will incrementally add features from:

1. `torch.nn`
2. `torch.optim`
3. `Dataset`
4. `DataLoader`

We will add one module at a time, showing exactly what each module does, and how it makes the code either more concise, or more flexible.

This tutorial assumes you already have PyTorch installed, and are familiar with the basics of tensor operations. (If you're familiar with Numpy array operations, you'll find the PyTorch tensor operations used here nearly identical).

Exercise 0: Library loading and installation

In case you don't have installed in your local computer the torch, numpy and scikit-learn packages, you can install them by running the following commands:

```
!pip install numpy
!pip install torch
!pip install scikit-learn
```

In [1]:
```
%matplotlib inline
!pip install numpy
!pip install torch
!pip install scikit-learn
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(1.25.2)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages
(2.2.1+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages
(from torch) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.
10/dist-packages (from torch) (4.10.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (f
rom torch) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages
(from torch) (3.2.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch) (3.1.3)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
(from torch) (2023.6.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in /usr/local/lib/
python3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105 in /usr/local/li
b/python3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in /usr/local/lib/
python3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cudnn-cu12==8.9.2.26 in /usr/local/lib/pytho
n3.10/dist-packages (from torch) (8.9.2.26)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in /usr/local/lib/pyth
on3.10/dist-packages (from torch) (12.1.3.1)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in /usr/local/lib/pyth
on3.10/dist-packages (from torch) (11.0.2.54)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in /usr/local/lib/py
thon3.10/dist-packages (from torch) (10.3.2.106)
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in /usr/local/lib/
python3.10/dist-packages (from torch) (11.4.5.107)
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in /usr/local/lib/
python3.10/dist-packages (from torch) (12.1.0.106)
Requirement already satisfied: nvidia-nccl-cu12==2.19.3 in /usr/local/lib/python3.
10/dist-packages (from torch) (2.19.3)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in /usr/local/lib/python
3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: triton==2.2.0 in /usr/local/lib/python3.10/dist-pac
kages (from torch) (2.2.0)
Requirement already satisfied: nvidia-nvjitlink-cu12 in /usr/local/lib/python3.10/
dist-packages (from nvidia-cusolver-cu12==11.4.5.107->torch) (12.4.99)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-p
ackages (from jinja2->torch) (2.1.5)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-pack
ages (from sympy->torch) (1.3.0)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-pack
ages (1.2.2)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-pac
kages (from scikit-learn) (1.25.2)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-pack
ages (from scikit-learn) (1.11.4)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-pac
kages (from scikit-learn) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/d
ist-packages (from scikit-learn) (3.3.0)
```

Exercise1: Digits Prediction

In this exercise we will train a simple logistic regression model to classify the digits in the **digits** dataset, a simplified version of the mnist dataset provided by scikit-learn. It consists of black-and-white images of hand-drawn digits (between 0 and 9).

Exercise 1.1: Data Loading and Visualization

Similarly, to the previous lab, we will download it as a dataframe from scikit-learn.

In [2]:

```
from sklearn.datasets import load_digits
import numpy as np
import torch

### ENTER YOUR CODE HERE (3-4 lines expected)
x, y = load_digits(return_X_y=True, as_frame=True)

n_samples, n_features = x.shape
n_classes = len(np.unique(y))

print(f"Number of samples: {n_samples}, Number of features: {n_features}")
print(f"Labels range {y.min()} - {y.max()}, Number of classes: {n_classes}")
```

```
Number of samples: 1797, Number of features: 64
Labels range 0 - 9, Number of classes: 10
```

Let's now visualize few statistics by using the `describe` method of the dataframe.

In [3]:

```
### ENTER YOUR CODE HERE (1 line expected)
x.describe()
```

Out[3]:

	pixel_0_0	pixel_0_1	pixel_0_2	pixel_0_3	pixel_0_4	pixel_0_5	pixel_0_6
count	1797.0	1797.000000	1797.000000	1797.000000	1797.000000	1797.000000	1797.000000
mean	0.0	0.303840	5.204786	11.835838	11.848080	5.781859	1.362270
std	0.0	0.907192	4.754826	4.248842	4.287388	5.666418	3.325775
min	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.0	0.000000	1.000000	10.000000	10.000000	0.000000	0.000000
50%	0.0	0.000000	4.000000	13.000000	13.000000	4.000000	0.000000
75%	0.0	0.000000	9.000000	15.000000	15.000000	11.000000	0.000000
max	0.0	8.000000	16.000000	16.000000	16.000000	16.000000	16.000000

8 rows × 64 columns

Do you already have any guess of which is the **least discriminative** feature? Also, shall we normalize the dataset?

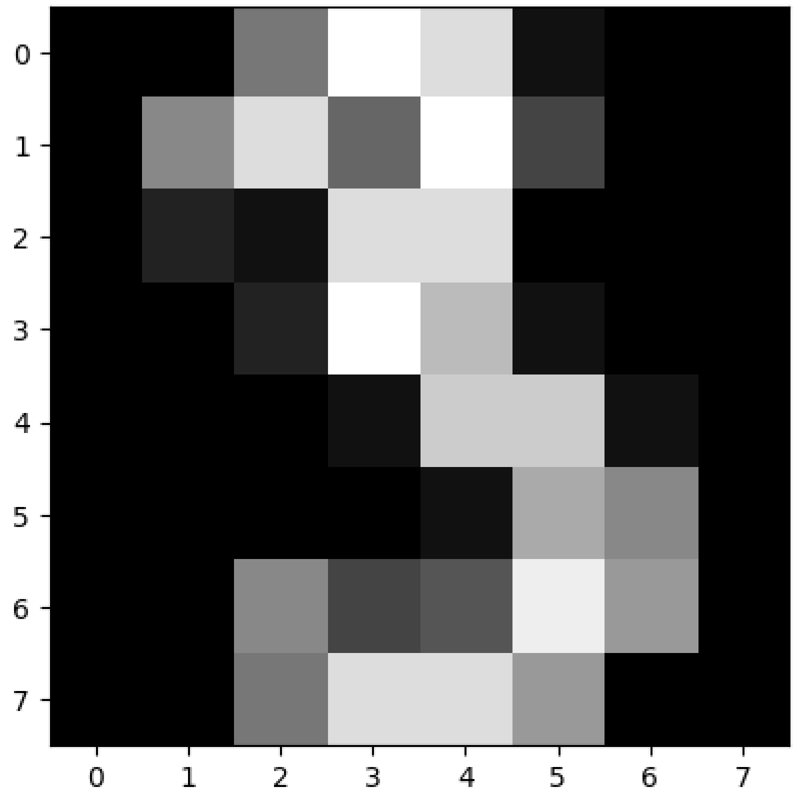
The pixels in the corners are surely the least discriminative, since they never change within the dataset images (mean = 0, std = 0)

Each image is 8 x 8, and is being stored as a flattened row of length 64. Let's take a look at one; we need to reshape it to 2d first. Then we will use the `imshow` function from matplotlib to visualize it.

```
In [4]: from matplotlib import pyplot as plt
        import numpy as np

        plt.imshow(x.iloc[3].values.reshape((8, 8)), cmap="gray")
        print(x.shape)
```

```
(1797, 64)
```



Exercise 1.2: Data Preprocessing

We have to split the dataset into training and validation. To do so we can use one of the many functions provided by scikit-learn. For example we can call `sklearn.model_selection.StratifiedKFold()` to have a cross validation generator.

For the sake of simplicity, however, in this tutorial we will use `sklearn.model_selection.train_test_split` which performs a stratified holdout cross-validation to create a single train test split (with 90% - 10% of data in this case)

```
In [5]: ### ENTER YOUR CODE HERE (2 lines expected)
        from sklearn.model_selection import train_test_split
        x_train, x_test, y_train, y_test = train_test_split(x.values, y.values, test_size=
```

I hope you positively answered the previous normalization question because yes, we **always need to normalize the data with neural networks**. The input data is stored with values in [0-16], we will normalize to [0-1].

$$x = \frac{x - min(x)}{max(x)}$$

We will use scikit-learn to do so, with the `MinMaxScaler` class. Remember to fit the scaler only on the training data, and then apply it to both the training and validation data.

In [6]:
```
from sklearn.preprocessing import MinMaxScaler

### ENTER YOUR CODE HERE (3 lines expected)
normalizer = MinMaxScaler()
x_train = normalizer.fit_transform(x_train)
x_test = normalizer.transform(x_test)

print(f"Train min: {x_train.min()}, max:{x_train.max()}")
print(f"Test min: {x_test.min()}, max:{x_test.max()}")
```

```
Train min: 0.0, max:1.0
Test min: 0.0, max:1.0
```

PyTorch uses `torch.tensor`, rather than numpy arrays, so we need to convert our data.

In [7]:
```
x_train = torch.as_tensor(x_train, dtype=torch.float32)
x_test = torch.as_tensor(x_test, dtype=torch.float32)
y_train = torch.as_tensor(y_train, dtype=torch.long)
y_test = torch.as_tensor(y_test, dtype=torch.long)
```

Exercise 1.3 Neural networks from scratch (no torch.nn)

Let's first create a model using nothing but PyTorch tensor operations. We assume you're already familiar with the basics of neural networks.

1. PyTorch provides methods to create random or zero-filled tensors, which we will use to create our weights and bias for a simple linear model.
2. These are just regular tensors, with one very special addition: we tell PyTorch that they require a **gradient**. This causes PyTorch to record all the operations done on the tensor.
3. This allows us and PyTorch to calculate the gradient during back-propagation **automatically**!

The Model

We will use a very simple logistic regression with 10 output nodes (as many as the number of classes).

For the weights, we set `requires_grad` after the initialization, since we don't want that step included in the gradient.

To Note:

1. We are initializing the weights here with the Xavier initialisation, by multiplying with 1/sqrt(n_features)).
2. The trailing `_` in PyTorch signifies that the operation is performed in-place (`requires_grad_()`)

In [8]:
```
# initialize weights and bias
weights = torch.randn(n_features, n_classes)
weights = weights / np.sqrt(n_features)
bias = torch.randn(n_classes)
bias = bias / np.sqrt(n_features)
```

```
# tell pytorch it requires to compute the gradient
weights.requires_grad_()
bias.requires_grad_()
```

Out[8]:
```
tensor([ 0.0834, -0.0747, -0.0608, -0.1283,  0.0189, -0.0840, -0.0916, -0.0156,
        -0.0880, -0.0045], requires_grad=True)
```

Thanks to PyTorch's ability to calculate gradients automatically, we can use any standard Python function (or callable object) as a model!

Let's just write a plain **matrix multiplication** and **broadcasted addition** to create a simple **linear model**.

We also need an activation function, we'll use a `log_softmax`. Although PyTorch provides lots of pre-written loss functions, activation functions, and so forth, you can easily write your own using plain python. PyTorch will even create fast GPU or vectorized CPU code for your function **automatically**.

In [9]:
```
def model(xb):
    return log_softmax(xb @ weights + bias) # a @ b -> torch.matmul(a, b)

def log_softmax(x):
    return x - x.exp().sum(-1).log().unsqueeze(-1) # => log(exp(x)/sum(exp(x_i)))
```

In the above, the `@` stands for the *matrix multiplication* operation.

Let's now call our model function on one batch of data (in this case, 64 images). This is one *forward pass*.

Note: our predictions won't be any better than random at this stage, since we start with random weights and we have not done any training step.

In [10]:
```
bs = 64               # batch size

### ENTER YOUR CODE HERE (2 lines expected)
xb = x_train[0:bs]    # a mini-batch from x
preds = model(xb)     # predictions

print(preds[0], preds.shape)
```

```
tensor([-1.9487, -2.5368, -1.6858, -2.2104, -3.0029, -1.8503, -3.5456, -2.3315,
        -2.2189, -3.1727], grad_fn=<SelectBackward0>) torch.Size([64, 10])
```

As you see, the `preds` tensor contains not only the tensor values, but also a gradient function. We'll use this later to do the backpropagation.

Loss and Metric

Let's implement negative log-likelihood to use as the loss function (again, we can just use standard Python).

Note that the way in which we have implemented it is just a clever way of doing the selection of the prediction corresponding to the correct label as in: $\sum_i y_i * log(f_i)$.

You can find further information on how to compute the cross entropy loss in different scenarios in this tutorial.

If we check our loss with our random model now, we can see if we improve after a backprop pass later.

```
In [11]: def nll(output, target):
             return -output[range(target.shape[0]), target].mean()

         loss_func = nll

         ### ENTER YOUR CODE HERE (1 line expected)
         l = loss_func(model(x_train), y_train)

         print(f"Initial Loss: {l}")
```

```
Initial Loss: 2.4288463592529297
```

Let's also implement a function to calculate the accuracy of our model.

For each prediction, if the index with the largest value matches the target value, then the prediction was correct.

Let's check the accuracy of our random model, so we can see if our accuracy improves as our loss improves.

```
In [12]: def accuracy(out, yb):

             ### ENTER YOUR CODE HERE (2 lines expected)
             preds = torch.argmax(out, dim=1)
             return (preds == yb).float().mean() * 100

         acc = accuracy(model(x_train), y_train)
         print(f"Initial Accuracy: {acc:.2f} %")
```

```
Initial Accuracy: 2.97 %
```

The training loop

We can now run a training loop. For each iteration, we will:

1. select a mini-batch of data (of size `bs`)
2. use the model to make predictions
3. calculate the loss
4. `loss.backward()` computes the gradients of the model, in this case, `weights` and `bias`.
5. update the parameters to optimize the model

Note:

- We do this within the `torch.no_grad()` context manager, because we do not want these actions to be recorded for our next calculation of the gradient. You can read more about how PyTorch's Autograd records operations here.
- We then set the gradients to zero, so that we are ready for the next loop. Otherwise, our gradients would record a running tally of all the operations that had happened (i.e. `loss.backward()` *adds* the gradients to whatever is already stored, rather than replacing them).

In [13]:
```
lr = .5  # learning rate
epochs = 20  # how many epochs to train for
train_samples = x_train.shape[0]

for epoch in range(epochs):
    for i in range((train_samples - 1) // bs + 1):

        ## select input and label batch
        start_i = i * bs
        end_i = start_i + bs
        xb = x_train[start_i:end_i]
        yb = y_train[start_i:end_i]

        ## make predictions and compute loss
        pred = model(xb)
        loss = loss_func(pred, yb)

        ## compute gradient through backprop
        loss.backward()

        ## update gradients through SGD
        with torch.no_grad():
            weights -= weights.grad * lr
            bias -= bias.grad * lr
            weights.grad.zero_()
            bias.grad.zero_()
    print(f"Epoch {epoch}: {loss_func(model(x_train), y_train):.2f}")
```

```
Epoch 0: 0.9641714096069336
Epoch 1: 0.612633466720581
Epoch 2: 0.4707716107368469
Epoch 3: 0.3937646746635437
Epoch 4: 0.34467172622680664
Epoch 5: 0.31020745635032654
Epoch 6: 0.2844368517398834
Epoch 7: 0.264294296503067
Epoch 8: 0.24802596867084503
Epoch 9: 0.23455022275447845
Epoch 10: 0.22316069900989532
Epoch 11: 0.21337513625621796
Epoch 12: 0.20485202968120575
Epoch 13: 0.19734247028827667
Epoch 14: 0.19066037237644196
Epoch 15: 0.18466360867023468
Epoch 16: 0.17924164235591888
Epoch 17: 0.17430725693702698
Epoch 18: 0.16979040205478668
Epoch 19: 0.16563434898853302
```

That's it: we've created and trained a minimal neural network (in this case, a logistic regression, since we have no hidden layers) entirely from scratch!

Let's check the loss and accuracy and compare those to what we got earlier. We expect that the loss will have decreased and accuracy to have increased, and they have.

In [18]:
```
### ENTER YOUR CODE HERE (2 lines expected)
print(f"Loss after training: {loss_func(model(x_train), y_train):.2f}")
print(f"Accuracy after training: {accuracy(model(x_train), y_train):.2f}")
```

```
Loss after training: 0.17
Accuracy after training: 96.60
```

Let's refactor our code using torch.nn

We will now refactor our code, so that it does the same thing as before, only we'll start taking advantage of PyTorch's `nn` classes to make it more concise and flexible.

At each step from here, we should be making our code one or more of:

1. shorter
2. more understandable
3. more flexible

Exercise 1.4 Refactor using nn.functional

The first and easiest step is to make our code shorter by replacing our hand-written activation and loss functions with those from `torch.nn.functional` (which is generally imported into the namespace `F` by convention).

The `torch.nn.functional` module contains all the functions in the `torch.nn` library (whereas other parts of the library contain classes). As well as a wide range of loss and activation functions, you'll also find here some convenient functions for creating deep neural networks, such as pooling functions.

If you're using negative log likelihood loss and log softmax activation, then Pytorch provides a single function `F.cross_entropy` that combines the two. So we can even remove the activation function from our model.

In [15]:
```
import torch.nn.functional as F

loss_func = F.cross_entropy

def model(xb):
    return xb @ weights + bias
```

Note that we no longer call `log_softmax` in the `model` function. Let's confirm that our loss and accuracy are the same as before:

In [17]:
```
### ENTER YOUR CODE HERE (2 line expected)
print(f"Loss after refactoring: {loss_func(model(x_train), y_train)}")
print(f"Accuracy after refactoring: {accuracy(model(x_train), y_train)}")
```

```
Loss after refactoring: 0.1656343787908554
Accuracy after refactoring: 96.59864044189453
```

Exercise 1.5 Refactor using nn.Module

Next up, we'll use `nn.Module` (*uppercase M*) and `nn.Parameter`, for a clearer and more concise training loop.

1. We subclass `nn.Module` (which itself is a class and able to keep track of its attributes). In this case, we want to create a class that holds our weights, bias, and method for the

forward step. `nn.Module` has a number of attributes and methods (such as `.parameters()` and `.zero_grad()`) which we will be using for this.
2. Since we're now using an object instead of just using a function, we first have to instantiate our model:

In [20]:
```
from torch import nn

class LogisticRegression(nn.Module):
    def __init__(self):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(n_features, n_classes) / np.sqrt(n_
        self.bias = nn.Parameter(torch.randn(n_classes) / np.sqrt(n_features))

    def forward(self, xb):
        return xb @ self.weights + self.bias

model = LogisticRegression()
```

Now we can calculate the loss in the same way as before. Note that `nn.Module` objects are used as if they are functions (i.e they are *callable*), but behind the scenes Pytorch will call our `forward` method automatically, i.e.

```
def __call__(self, *args, **kwargs):
  return self.forward(args, kwargs)
```

In [21]:
```
### ENTER YOUR CODE HERE (1 line expected)
print(loss_func(model(x_train), y_train), accuracy(model(x_train), y_train))
```

```
tensor(2.4198, grad_fn=<NllLossBackward0>) tensor(11.1317)
```

model.parameters() and model.zero_grad()

Previously for our training loop we had to update the values for each parameter by name, and manually zero out the grads for each parameter separately, like this:

```
with torch.no_grad():
    weights -= weights.grad * lr
    bias -= bias.grad * lr
    weights.grad.zero_()
    bias.grad.zero_()
```

Now we can take advantage of `model.parameters()` and `model.zero_grad()` to make those steps more concise and less prone to the error of forgetting some of our parameters, particularly if we had a more complicated model:

```
with torch.no_grad():
    for p in model.parameters():
        p -= p.grad * lr
    model.zero_grad()
```

We'll wrap our little training loop in a `fit` function so we can run it again later. And we double-check that our loss has gone down:

In [23]:
```
def fit():
    for epoch in range(epochs):
        for i in range((train_samples - 1) // bs + 1):
            start_i = i * bs
            end_i = start_i + bs
            xb = x_train[start_i:end_i]
            yb = y_train[start_i:end_i]
            pred = model(xb)
            loss = loss_func(pred, yb)

            loss.backward()

            ### ENTER YOUR CODE HERE (4 lines expected)
            with torch.no_grad():
                for p in model.parameters():
                    p -= p.grad * lr
                model.zero_grad()
        print(f"Epoch: {epoch}, {loss_func(model(x_train), y_train):.2f}")

fit()
print(f"Accuracy: {accuracy(model(x_train), y_train):.2f}")
```

```
Epoch: 0, 0.16
Epoch: 1, 0.16
Epoch: 2, 0.15
Epoch: 3, 0.15
Epoch: 4, 0.15
Epoch: 5, 0.15
Epoch: 6, 0.14
Epoch: 7, 0.14
Epoch: 8, 0.14
Epoch: 9, 0.14
Epoch: 10, 0.13
Epoch: 11, 0.13
Epoch: 12, 0.13
Epoch: 13, 0.13
Epoch: 14, 0.13
Epoch: 15, 0.13
Epoch: 16, 0.12
Epoch: 17, 0.12
Epoch: 18, 0.12
Epoch: 19, 0.12
Accuracy: 97.65
```

Exercise 1.6 Refactor using nn.Linear

We continue to refactor our code. Instead of manually defining and initializing `self.weights` and `self.bias`, and calculating `xb  @ self.weights + self.bias`, we will instead use the Pytorch class nn.Linear for a linear layer, which does all that for us.

Pytorch has many types of **predefined layers** that can greatly simplify our code, and often makes it faster too!

In [24]:
```
class LogisticRegression(nn.Module):
    def __init__(self):
        super().__init__()
        ### ENTER YOUR CODE HERE (1 line expected)
        self.lin = nn.Linear(n_features, n_classes)

    def forward(self, xb):
        ### ENTER YOUR CODE HERE (1 lines expected)
        return self.lin(xb)
```

We instantiate our model and calculate the loss in the same way as before:

In [25]:
```
model = LogisticRegression()
### ENTER YOUR CODE HERE (1 line expected)
```

```
Loss: 2.4040098190307617, Accuracy: 9.461966514587402
```

We are still able to use our same `fit` method as before.

In [26]:
```
fit()
print(f"Loss: {loss_func(model(x_train), y_train)}, Accuracy: {accuracy(model(x_tra
```

```
Epoch: 0, 0.98
Epoch: 1, 0.62
Epoch: 2, 0.47
Epoch: 3, 0.40
Epoch: 4, 0.35
Epoch: 5, 0.31
Epoch: 6, 0.29
Epoch: 7, 0.27
Epoch: 8, 0.25
Epoch: 9, 0.24
Epoch: 10, 0.22
Epoch: 11, 0.21
Epoch: 12, 0.21
Epoch: 13, 0.20
Epoch: 14, 0.19
Epoch: 15, 0.19
Epoch: 16, 0.18
Epoch: 17, 0.17
Epoch: 18, 0.17
Epoch: 19, 0.17
Loss: 0.16625221073627472, Accuracy: 96.78417205810547
```

Exercise 1.7 Refactor using optim

Pytorch also has a package with various optimization algorithms, `torch.optim`. To implement the previous stochastic gradient descent (SGD) optimization step, we can use `optim.SGD(params, lr)` passing to it all the model parameters and the learning rate.

We can use the `step` method from our optimizer to take a forward step, instead of manually updating each parameter.

This will let us replace our previous manually coded optimization step:

```
with torch.no_grad():
    for p in model.parameters():
        p -= p.grad * lr
    model.zero_grad()
```

and instead use just:

```
# before the training loop
opt = optim.SGD(model.parameters(), lr=lr)

# in the training loop
```

```
    opt.step()
    opt.zero_grad()
```

`optim.zero_grad()` resets the gradient to 0 and we need to call it **before** computing the gradient for the next minibatch.

In [28]:
```
from torch import optim
model = LogisticRegression()

### ENTER YOUR CODE HERE (1 line expected)
opt = optim.SGD(model.parameters(), lr=lr)

print(f"Accuracy: {accuracy(model(x_train), y_train)}")

def fit():
    for epoch in range(epochs):
        for i in range((train_samples - 1) // bs + 1):
            start_i = i * bs
            end_i = start_i + bs
            xb = x_train[start_i:end_i]
            yb = y_train[start_i:end_i]
            pred = model(xb)
            loss = loss_func(pred, yb)

            loss.backward()

            ### ENTER YOUR CODE HERE (2 lines expected)
            opt.step()
            opt.zero_grad()
        print(f"Epoch: {epoch}, {loss_func(model(x_train), y_train)}")

fit()
print(f"Accuracy: {accuracy(model(x_train), y_train)}")
```

```
Accuracy: 3.339517593383789
Epoch: 0, 0.9558985233306885
Epoch: 1, 0.6075210571289062
Epoch: 2, 0.4672076404094696
Epoch: 3, 0.3910726308822632
Epoch: 4, 0.3425308167934418
Epoch: 5, 0.30843910574913025
Epoch: 6, 0.2829316258430481
Epoch: 7, 0.2629814147949219
Epoch: 8, 0.24685810506343842
Epoch: 9, 0.2334948033094406
Epoch: 10, 0.2221948206424713
Epoch: 11, 0.2124822735786438
Epoch: 12, 0.2040199488401413
Epoch: 13, 0.19656190276145935
Epoch: 14, 0.18992413580417633
Epoch: 15, 0.18396596610546112
Epoch: 16, 0.17857804894447327
Epoch: 17, 0.17367388308048248
Epoch: 18, 0.1691841185092926
Epoch: 19, 0.1650523990392685
Accuracy: 96.72232055664062
```

Exercise 1.8 Refactor using Dataset

PyTorch has an abstract Dataset class in `torch.utils.data.Dataset`. A Dataset can be anything that has:

- a `__len__` function (called by Python's standard `len` function)
- a `__getitem__` function as a way of indexing into it.

There are many pre-existing datasets in the torchvision library. Otherwise you can also create your own. This tutorial walks through a nice example of creating a custom `FacialLandmarkDataset` class as a subclass of `Dataset`.

PyTorch's TensorDataset is a Dataset wrapping tensors. By defining a length and way of indexing, this also gives us a way to iterate, index, and slice along the first dimension of a tensor.

Both `x_train` and `y_train` can be combined in a single `TensorDataset`, which will be easier to iterate over and slice.

In [29]:
```python
from torch.utils.data import TensorDataset
train_ds = TensorDataset(x_train, y_train)
```

Previously, we had to iterate through minibatches of x and y values separately:

```
xb = x_train[start_i:end_i]
yb = y_train[start_i:end_i]
```

Now, we can do these two steps together:

```
    xb,yb = train_ds[i*bs : i*bs+bs] # we access all tensors in the
dataset with one slicing
```

In [30]:
```python
model = LogisticRegression()
opt = optim.SGD(model.parameters(), lr=lr)

print(f"Accuracy: {accuracy(model(x_train), y_train)}")
def fit():
    for epoch in range(epochs):
        for i in range((train_samples - 1) // bs + 1):
            xb, yb = train_ds[i * bs: i * bs + bs]
            pred = model(xb)
            loss = loss_func(pred, yb)

            loss.backward()
            opt.step()
            opt.zero_grad()

        print(f"Epoch: {epoch}, {loss_func(model(x_train), y_train)}")

fit()
print(f"Accuracy: {accuracy(model(x_train), y_train)}")
```

```
Accuracy: 9.400123596191406
Epoch: 0, 0.957848846912384
Epoch: 1, 0.6084758639335632
Epoch: 2, 0.4680284559726715
Epoch: 3, 0.39175915718078613
Epoch: 4, 0.34309670329093933
Epoch: 5, 0.3089105784893036
Epoch: 6, 0.2833327353000641
Epoch: 7, 0.26333051919937134
Epoch: 8, 0.24716828763484955
Epoch: 9, 0.23377536237239838
Epoch: 10, 0.22245240211486816
Epoch: 11, 0.21272163093090057
Epoch: 12, 0.20424474775791168
Epoch: 13, 0.1967749148607254
Epoch: 14, 0.1901274174451828
Epoch: 15, 0.18416129052639008
Epoch: 16, 0.17876674234867096
Epoch: 17, 0.17385707795619965
Epoch: 18, 0.16936266422271729
Epoch: 19, 0.16522714495658875
Accuracy: 96.78417205810547
```

Exercise 1.9 Refactor using DataLoader

Pytorch's `DataLoader` is responsible for managing batches. You can create a `DataLoader` from any `Dataset`. `DataLoader` makes it easier to iterate over batches. Rather than having to use `train_ds[i*bs : i*bs+bs]`, the DataLoader gives us each minibatch automatically.

Also Shuffling the training data is important to prevent correlation between batches and overfitting. DataLoader provides the parameter `shuffle=True` to do it.

For data intensive training (e.g. big images, or videos), it also allows to employ multiprocessing to load the data in parallel to the training loop with the parameter `num_workers`.

In [31]:
```python
from torch.utils.data import DataLoader

train_ds = TensorDataset(x_train, y_train)
train_dl = DataLoader(train_ds, batch_size=bs, shuffle=True)
```

Previously, our loop iterated over batches (xb, yb) like this:

```
for i in range((n-1)//bs + 1):
    xb,yb = train_ds[i*bs : i*bs+bs]
    pred = model(xb)
```

Now, our loop is much cleaner, as (xb, yb) are loaded automatically from the data loader:

```
for xb,yb in train_dl:
    pred = model(xb)
```

In [32]:
```python
model = LogisticRegression()
opt = optim.SGD(model.parameters(), lr=lr)
```

```python
print(f"Accuracy: {accuracy(model(x_train), y_train)}")
def fit():
    for epoch in range(epochs):

        ### ENTER YOUR CODE HERE (2 lines expected)
        for xb, yb in train_dl: # we don't have to specify anymore nor bs nor trair
            pred = model(xb)
            loss = loss_func(pred, yb)

            loss.backward()
            opt.step()
            opt.zero_grad()

        print(f"Epoch: {epoch}, {loss_func(model(x_train), y_train)}")

fit()
print(f"Accuracy: {accuracy(model(x_train), y_train)}")
```

```
Accuracy: 6.9264068603515625
Epoch: 0, 1.0679187774658203
Epoch: 1, 0.6330335140228271
Epoch: 2, 0.4858763813972473
Epoch: 3, 0.3934139311313629
Epoch: 4, 0.35309505462646484
Epoch: 5, 0.3109516501426697
Epoch: 6, 0.2934253513813019
Epoch: 7, 0.27343761920928955
Epoch: 8, 0.25147292017936707
Epoch: 9, 0.23902444541454315
Epoch: 10, 0.23006980121135712
Epoch: 11, 0.2136516571044922
Epoch: 12, 0.20626300573349
Epoch: 13, 0.20276904106140137
Epoch: 14, 0.195303812623024
Epoch: 15, 0.18514080345630646
Epoch: 16, 0.17940817773342133
Epoch: 17, 0.1785695105791092
Epoch: 18, 0.1706557273864746
Epoch: 19, 0.16499225795269012
Accuracy: 96.66048431396484
```

Thanks to Pytorch's `nn.Module`, `nn.Parameter`, `Dataset`, and `DataLoader`, our training loop is now dramatically smaller and easier to understand. Let's now try to add the basic features necessary to create effective models in practice.

Exercise 1.10 Add validation

So far, we were just trying to get a reasonable training loop set up for use on our training data. In reality, you **always** should also have a validation set, in order to identify if you are overfitting. We will extract it from the training set and use it to evaluate the model at the end of each epoch.

We'll use a batch size for the test set that is twice as large as that for the training set. This is because the validation set does not need backpropagation and thus takes less memory (it doesn't need to store the gradients). We take advantage of this to use a larger batch size and compute the loss more quickly by inserting the validation loop in a `with torch.no_grad():`.

In [33]:
```
x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train, test_size=0

train_ds = TensorDataset(x_train, y_train)
train_dl = DataLoader(train_ds, batch_size=bs, shuffle=True)

valid_ds = TensorDataset(x_valid, y_valid)
valid_dl = DataLoader(valid_ds, batch_size=bs * 2) # No need to shuffle the Validat
```

We will calculate and print the validation loss at the end of each epoch.

Note that we always call `model.train()` before training, and `model.eval()` before inference, because these are used by some layers (not employed here) such as `nn.BatchNorm2d` and `nn.Dropout` to ensure appropriate behaviour for these different phases.

In [36]:
```
model = LogisticRegression()
opt = optim.SGD(model.parameters(), lr=lr)

def fit():
    for epoch in range(epochs):
        ### ENTER YOUR CODE HERE (1 line expected)
        model.train()

        for xb, yb in train_dl:
            pred = model(xb)
            loss = loss_func(pred, yb)

            loss.backward()
            opt.step()
            opt.zero_grad()


        ### ENTER YOUR CODE HERE (6-8 lines expected)
        model.eval()
        with torch.no_grad():
            valid_losses, valid_accs = [], []
            for xb, yb in valid_dl:
                valid_losses.append(loss_func(model(xb), yb))
                valid_accs.append(accuracy(model(xb), yb))

        print(f"Epoch {epoch:.2f}: Val Loss {np.mean(valid_losses):.2f}, "
              f"Val Acc: {np.mean(valid_accs):.2f}")

fit()
```

```
Epoch 0.00: Val Loss 1.10, Val Acc: 89.73
Epoch 1.00: Val Loss 0.72, Val Acc: 90.03
Epoch 2.00: Val Loss 0.57, Val Acc: 91.98
Epoch 3.00: Val Loss 0.49, Val Acc: 88.26
Epoch 4.00: Val Loss 0.43, Val Acc: 90.12
Epoch 5.00: Val Loss 0.40, Val Acc: 92.76
Epoch 6.00: Val Loss 0.37, Val Acc: 89.43
Epoch 7.00: Val Loss 0.34, Val Acc: 91.29
Epoch 8.00: Val Loss 0.32, Val Acc: 93.15
Epoch 9.00: Val Loss 0.31, Val Acc: 93.15
Epoch 10.00: Val Loss 0.29, Val Acc: 94.62
Epoch 11.00: Val Loss 0.28, Val Acc: 95.01
Epoch 12.00: Val Loss 0.28, Val Acc: 92.76
Epoch 13.00: Val Loss 0.27, Val Acc: 93.93
Epoch 14.00: Val Loss 0.26, Val Acc: 94.62
Epoch 15.00: Val Loss 0.25, Val Acc: 93.15
Epoch 16.00: Val Loss 0.25, Val Acc: 93.93
Epoch 17.00: Val Loss 0.24, Val Acc: 95.40
Epoch 18.00: Val Loss 0.23, Val Acc: 94.32
Epoch 19.00: Val Loss 0.24, Val Acc: 95.01
```

Exercise 1.11 Saving best model: early stopping

It may have happend that the model has reached the highest validation accuracy (and/or lower validation loss) not at the last epoch. It means that the model overfitted the training data.

A possible way to avoid this phenomenon is to save the model achieving the best validation accuracy (other possible solutions includes reducing the learning rate and decreasing the number of training epochs).

There exists two possible way to do it in pytorch. `torch.save()` and `torch.load()` can be used with any kind of objects. Torch will serialized this object through pickle. However, if the code generating the object is modified the code might brake in several ways. To avoid this issue, torch provides another couple of functions, `model.state_dict()` and `model.load_state_dict()`, which only save and load the weights of the network.

```
torch.save(model.state_dict(), "./best_model.pt")
model.load_state_dict(torch.load("./best_model.pt"))
```

For further information on this topic, please read this tutorial.

In [37]:
```
model = LogisticRegression()
opt = optim.SGD(model.parameters(), lr=lr)

def fit():
    best_acc = 0.
    for epoch in range(epochs):
        model.train()
        for xb, yb in train_dl:
            pred = model(xb)
            loss = loss_func(pred, yb)

            loss.backward()
            opt.step()
            opt.zero_grad()
        model.eval()
        with torch.no_grad():
            valid_losses, valid_accs = [], []
            for xb, yb in valid_dl:
                valid_losses.append(loss_func(model(xb), yb))
                valid_accs.append(accuracy(model(xb), yb))

            if best_acc < np.mean(valid_accs):

                ### ENTER YOUR CODE HERE - SAVE MODEL (1 line expected)
                torch.save(model.state_dict(), "./best_model.pt")

                best_acc = np.mean(valid_accs)

        print(f"Epoch {epoch}: Loss {np.mean(valid_losses):.2f},  \
                Acc: {np.mean(valid_accs):.2f}")


    ### ENTER YOUR CODE HERE (2 lines expected)
    model.load_state_dict(torch.load("./best_model.pt"))

    model.eval()
    print(f"Best Valid Acc: {np.mean([accuracy(model(xb), yb) for xb, yb in valid_c

fit()
```

```
Epoch 0: Loss 1.05,                     Acc: 88.47
Epoch 1: Loss 0.71,                     Acc: 91.20
Epoch 2: Loss 0.58,                     Acc: 90.81
Epoch 3: Loss 0.48,                     Acc: 91.59
Epoch 4: Loss 0.43,                     Acc: 90.51
Epoch 5: Loss 0.39,                     Acc: 89.04
Epoch 6: Loss 0.36,                     Acc: 91.29
Epoch 7: Loss 0.34,                     Acc: 94.62
Epoch 8: Loss 0.33,                     Acc: 92.76
Epoch 9: Loss 0.31,                     Acc: 93.54
Epoch 10: Loss 0.30,                     Acc: 92.76
Epoch 11: Loss 0.28,                     Acc: 92.76
Epoch 12: Loss 0.28,                     Acc: 93.15
Epoch 13: Loss 0.27,                     Acc: 92.76
Epoch 14: Loss 0.26,                     Acc: 93.93
Epoch 15: Loss 0.25,                     Acc: 92.76
Epoch 16: Loss 0.25,                     Acc: 93.93
Epoch 17: Loss 0.24,                     Acc: 93.54
Epoch 18: Loss 0.24,                     Acc: 93.93
Epoch 19: Loss 0.24,                     Acc: 92.76
Best Valid Acc: 94.62
```

Was it useful to early stop the training? Is the best accuracy the same as the one at the last epoch?

Finally, we test the model on the test set to see if it generalizes well also there. **Note**: There might be a discrepancy between the validation and test accuracy, but it should **not** be too large.

In [39]:
```
model = model.cpu()

### ENTER YOUR CODE HERE (2 lines expected)
test_dl = DataLoader(TensorDataset(x_test, y_test), batch_size=bs * 2)
print(f"Test Acc: {np.mean([accuracy(model(xb), yb) for xb, yb in test_dl]):.2f}")
```

```
Test Acc: 95.13
```

Exercise 1.12 Using the GPU

If you're lucky enough to have access to a CUDA-capable GPU you can use it to speed up your code. To do so in COLAB you can change *runtime type* in the *Runtime* dropdown menu to enable GPU computation. After changing the runtime type you will need to rerun the whole notebook beacause all variables will be lost (*Runtime* -> *Run all*).

Let's check that your GPU is working with Pytorch. It should print `True` if everything is set up correctly.

In [38]:
```
print(torch.cuda.is_available())
```

```
True
```

Let's then create a device object for it:

In [40]:
```
dev = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Finally, we can move our model and our data to the GPU. Generally dataset do not fit in the GPU, so we need to move only the batches inside the training/validation loops:

```
xb = xb.to(dev)
yb = yb.to(dev)
```

In [ ]:
```
model = LogisticRegression().to(dev)
opt = optim.SGD(model.parameters(), lr=lr)

def fit():
    best_acc = 0.
    for epoch in range(epochs):
        model.train()
        for xb, yb in train_dl:
            ### ENTER YOUR CODE HERE (1 line expected)
            xb, yb = xb.to(dev), yb.to(dev)
            pred = model(xb)
            loss = loss_func(pred, yb)

            loss.backward()
            opt.step()
            opt.zero_grad()

        model.eval()
        with torch.no_grad():
            valid_losses, valid_accs = [], []
            for xb, yb in valid_dl:
                ### ENTER YOUR CODE HERE (1 line expected)
                xb, yb = xb.to(dev), yb.to(dev)
                valid_losses.append(loss_func(model(xb), yb).cpu())
                valid_accs.append(accuracy(model(xb), yb).cpu())

            if best_acc < np.mean(valid_accs):
                torch.save(model.state_dict(), "./best_model.pt")
                best_acc = np.mean(valid_accs)

        print(f"Epoch {epoch}: Loss {np.mean(valid_losses):.2f}, "
              f"Acc: {np.mean(valid_accs):.2f}")
    model.load_state_dict(torch.load("./best_model.pt"))
    model.eval()
    print(f"Best Valid Acc: {np.mean([accuracy(model(xb.to(dev)), yb.to(dev)).cpu(

fit()
```

You should find it runs faster now

Exercise 2 What has the network learnt?

So far we have seen that the network has learnt, but how can we visualize it? There exists several mechanism to do so.

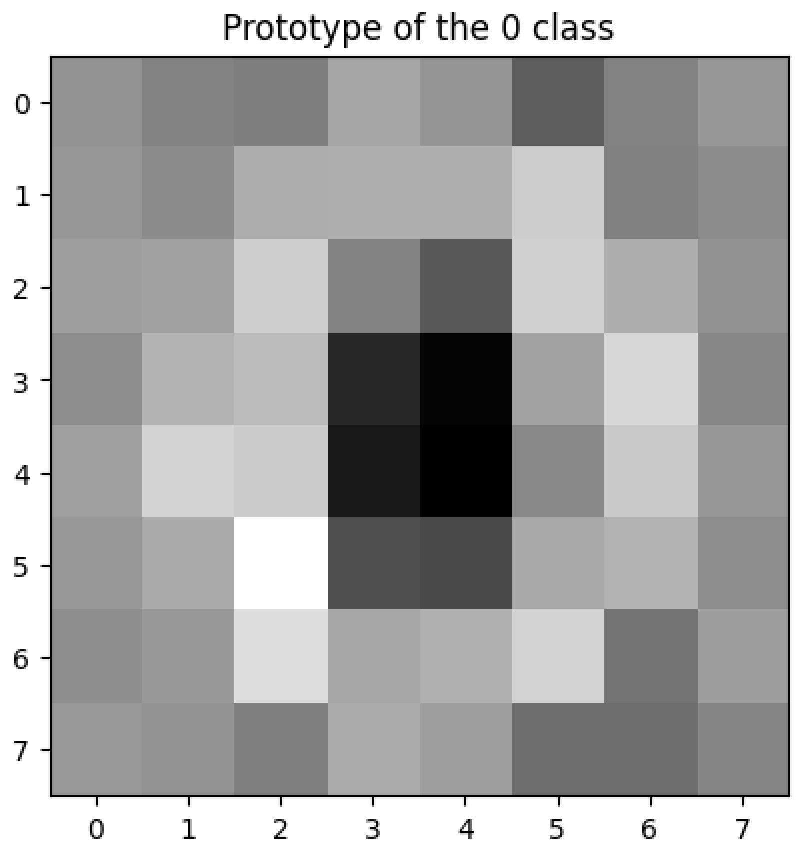
Exercise 2.1 Weight visualization

On a logistic regression we can directly visualize both biases and weights! We will reuse the same `im.show()` function that we used before to visualize the learnt weights.

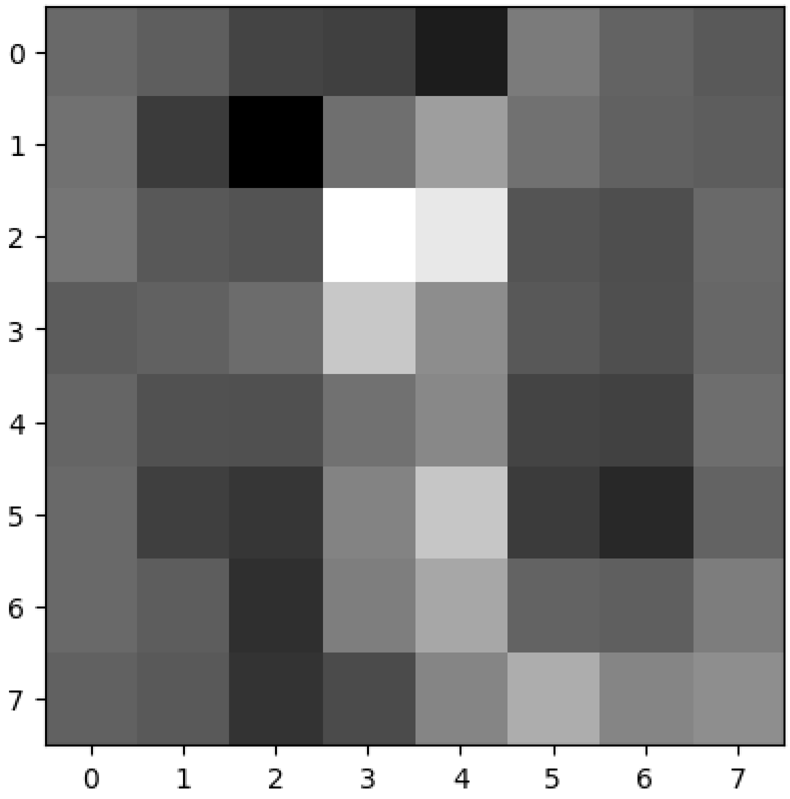
In [41]:
```
learnt_bias = model.lin.bias.cpu().detach()
learnt_weights = model.lin.weight.cpu().detach()
print(f"Bias: {learnt_bias}")
for i in range(10):
  ### ENTER YOUR CODE HERE (1 line expected)
  plt.imshow(learnt_weights[i].reshape((8, 8)), cmap="gray")

  plt.title(f"Prototype of the {i} class")
  plt.show()
```

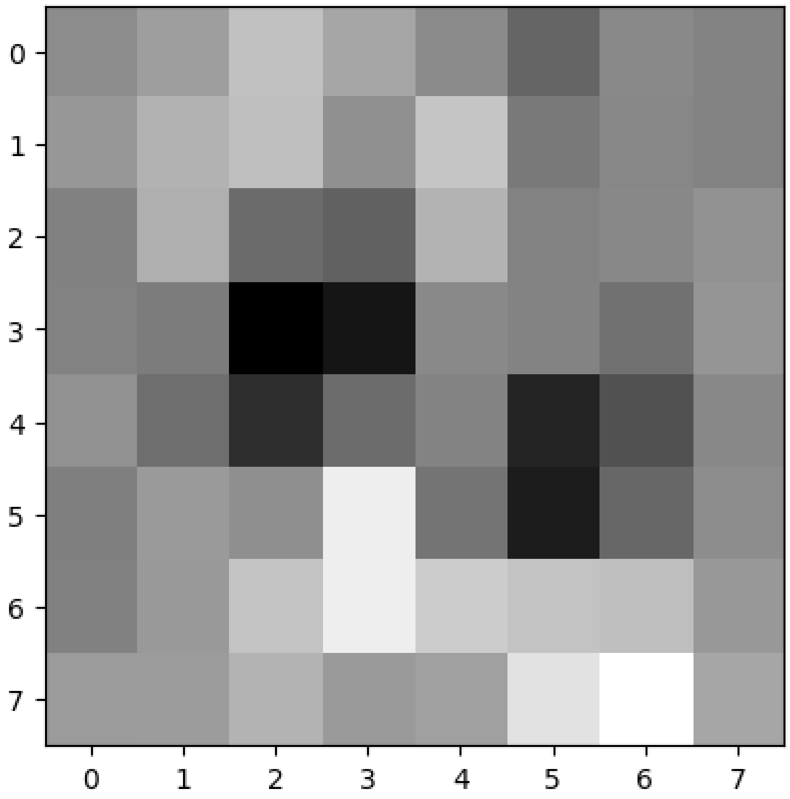
```
Bias: tensor([-0.1077, -0.0914,  0.1060,  0.0211, -0.0082, -0.0017, -0.0548,  0.08
36,
        -0.1162,  0.0490])
```



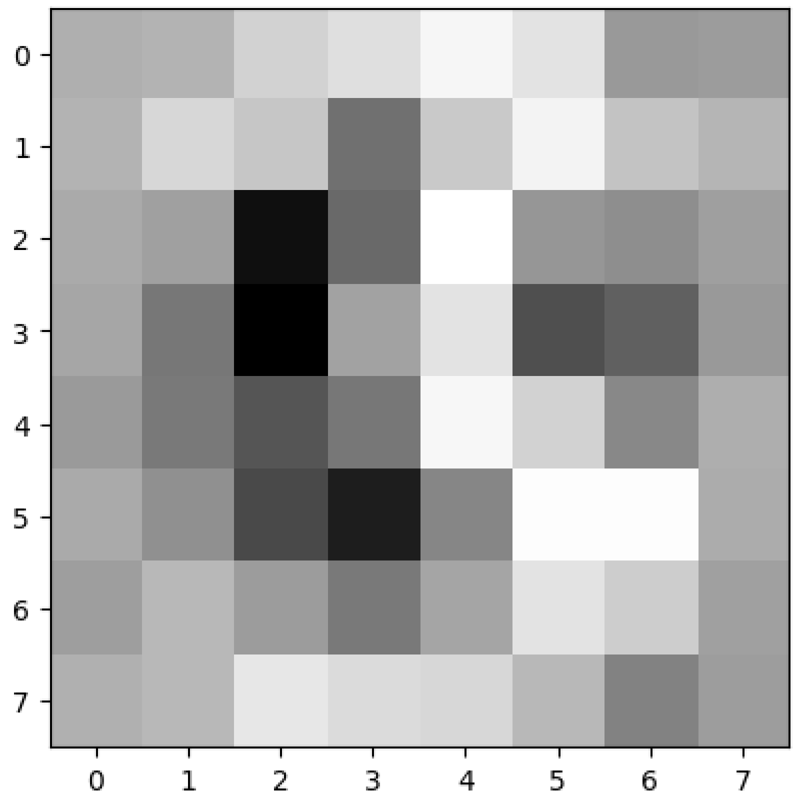
Prototype of the 1 class



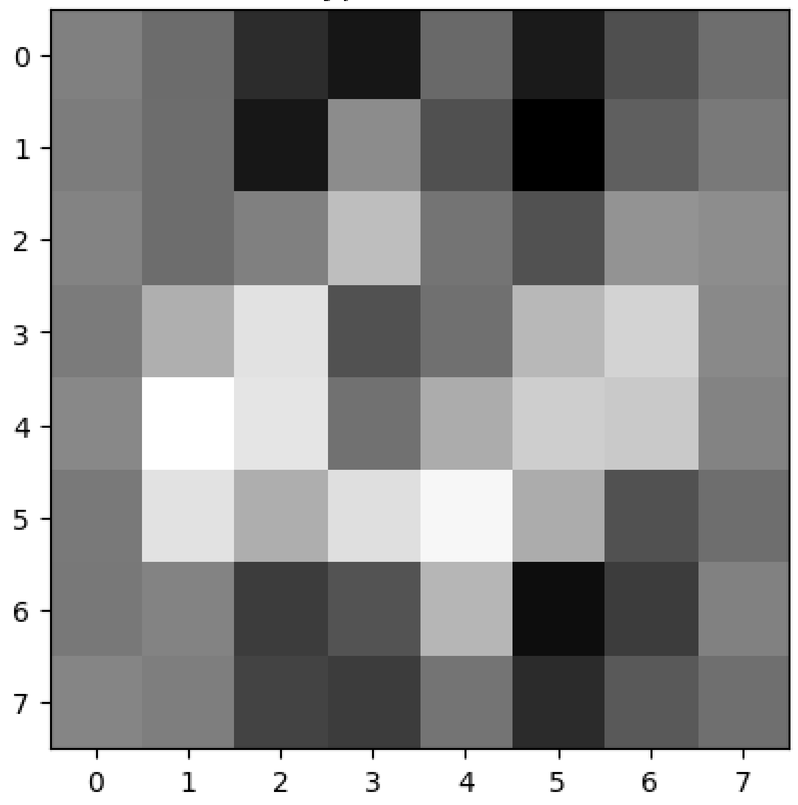
Prototype of the 2 class



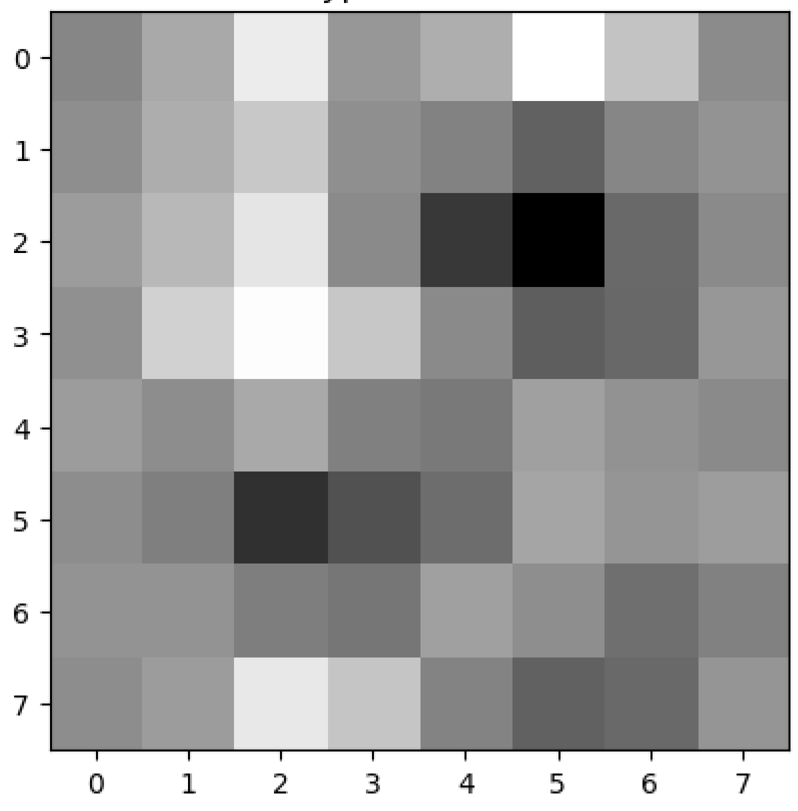
Prototype of the 3 class



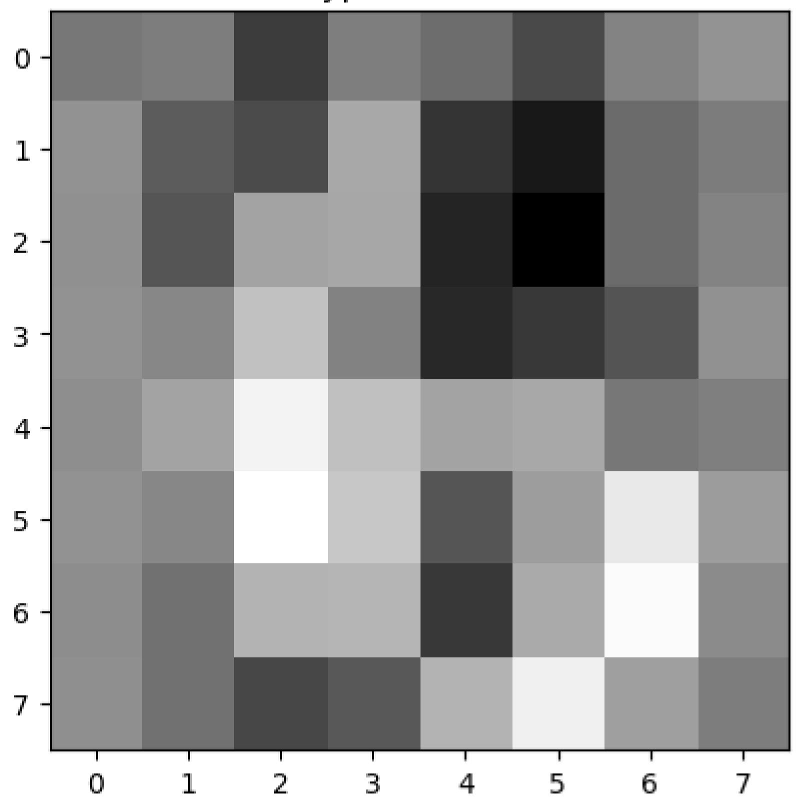
Prototype of the 4 class



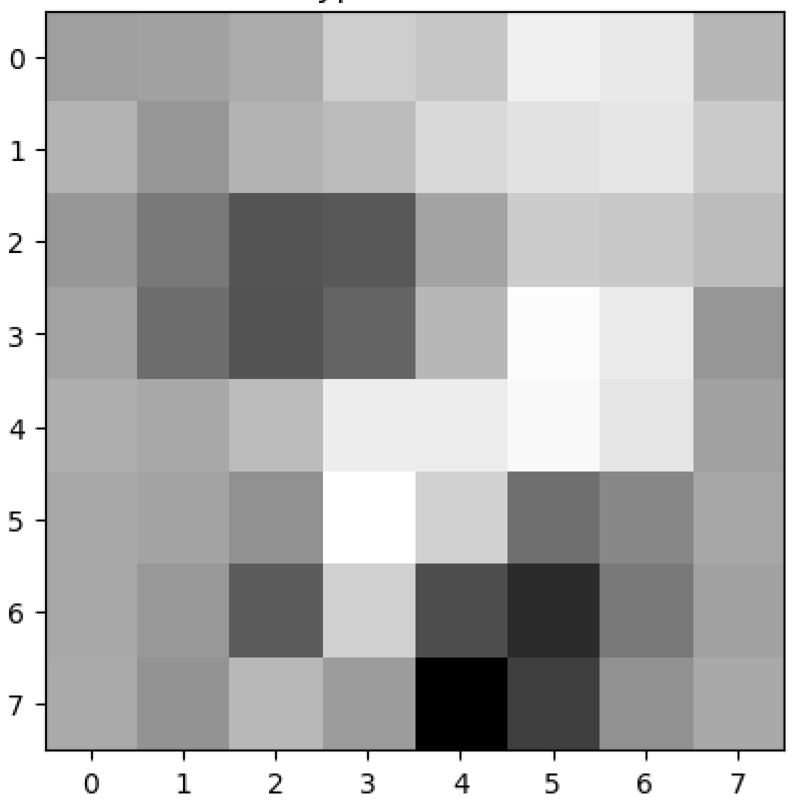
Prototype of the 5 class



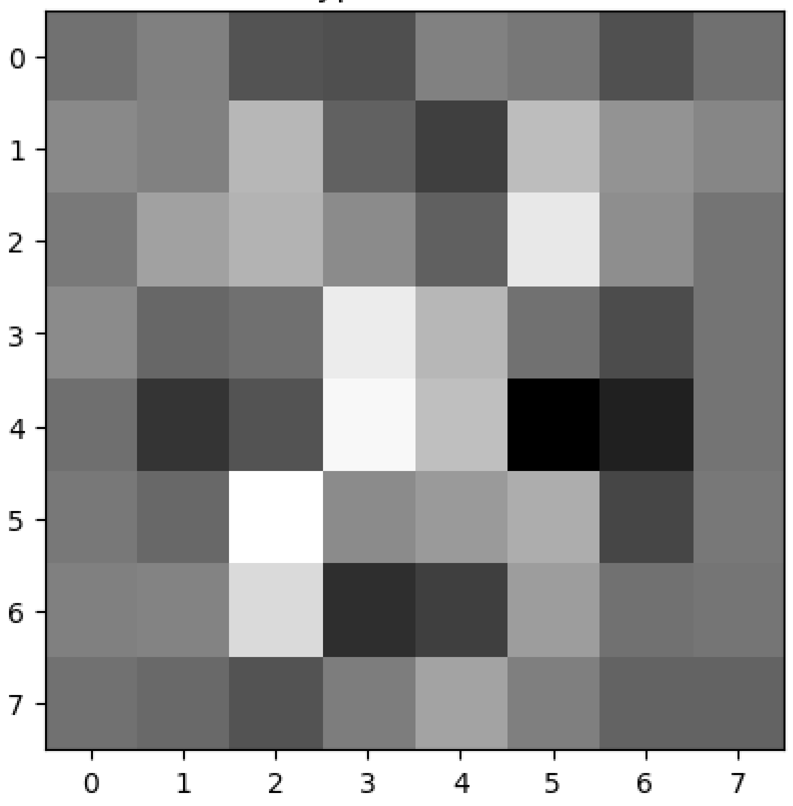
Prototype of the 6 class

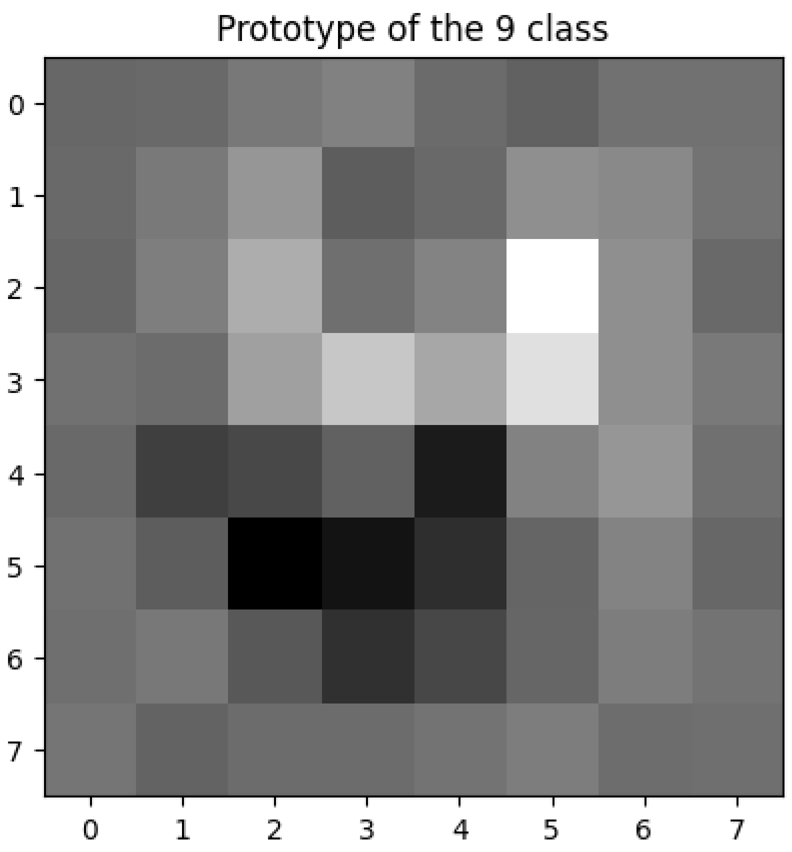


Prototype of the 7 class



Prototype of the 8 class





The images are surely blurry but they resemble somehow the standard patterns of the 0-9 digits

Exercise 2.1 T-SNE weight projection

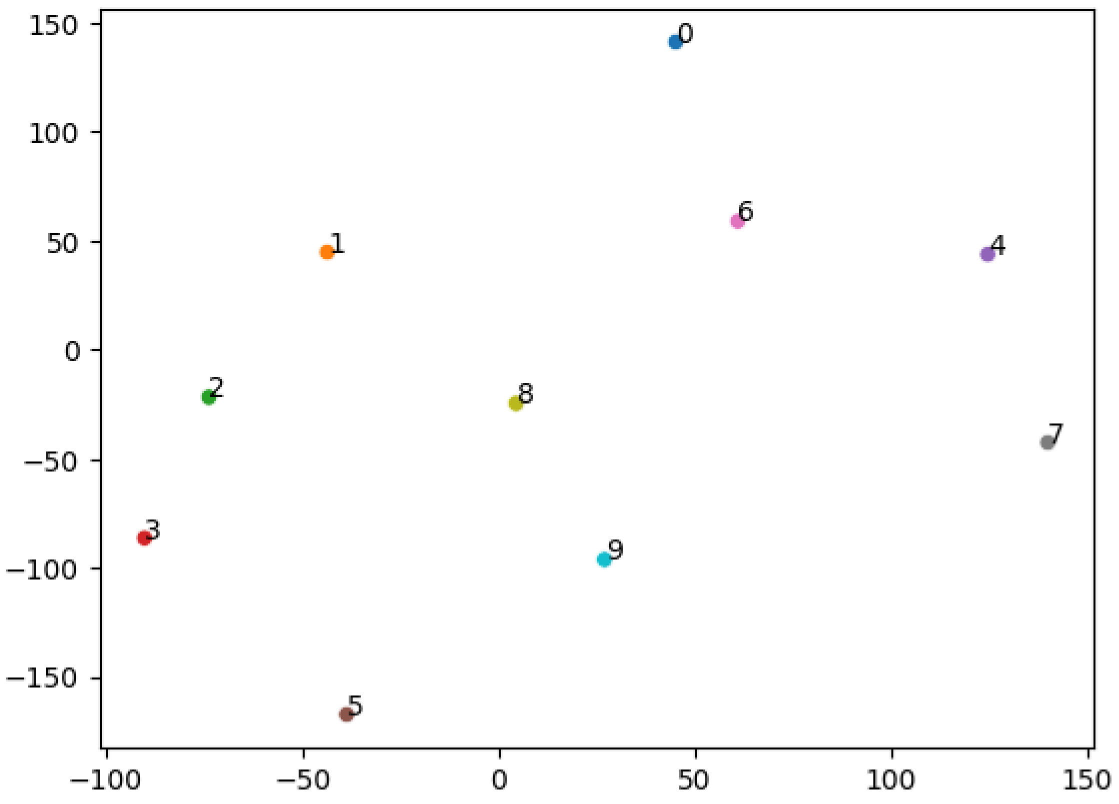
Otherwise with more complex model you can still see the distribution of the classes as learnt by the model through some visualization techniques. Personally I like very much t-sne: it allows to project into low dimensional representation (e.g. 2D) data which lies in very high dimensional spaces.

Let's try to projects the learnt weights into a 2D space.

In [50]:
```
import seaborn as sns
from sklearn.manifold import TSNE

projected_weights = TSNE(n_components=2,init="pca", learning_rate='auto', perplexit

sns.scatterplot(x=projected_weights[:,0], y=projected_weights[:,1], hue=(str(i) for
[plt.text(projected_weights[i,0], projected_weights[i,1], s=str(i)) for i in range(
plt.show()
```

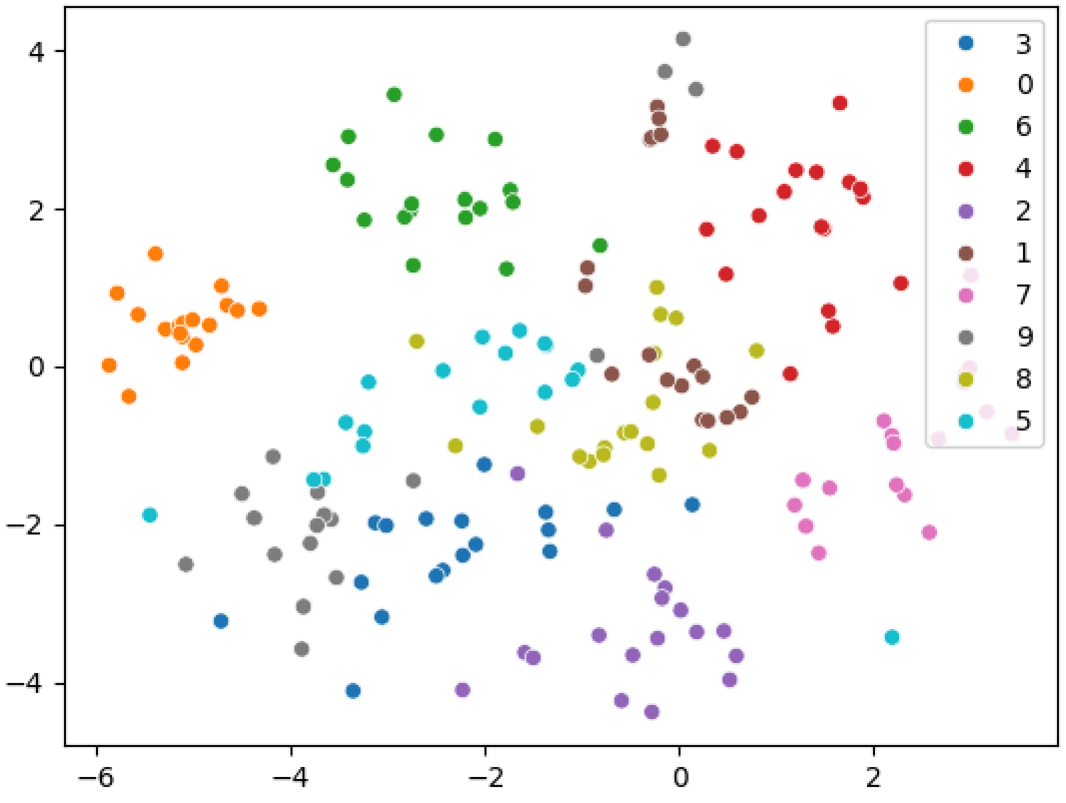


But you can do it with the samples as well! Let's do it with the samples in the test set

In [53]:
```
import seaborn as sns
from sklearn.manifold import TSNE

projected_samples = TSNE(n_components=2, init="pca", learning_rate='auto').fit_trar

sns.scatterplot(x=projected_samples[:,0], y=projected_samples[:,1], hue=[str(y_i.it
plt.show()
```



Closing thoughts

We now have a general data pipeline and training loop which you can use for training many types of models using Pytorch.

We promised at the start of this tutorial we'd explain through example each of `torch.nn`, `torch.optim`, `Dataset`, and `DataLoader`. So let's summarize what we've seen:

- **torch.nn**
  - `Module`: creates a callable which behaves like a function, but can also contain state(such as neural net layer weights). It knows what `Parameter` (s) it contains and can zero all their gradients, loop through them for weight updates, etc.
  - `Parameter`: a wrapper for a tensor that tells a `Module` that it has weights that need updating during backprop. Only tensors with the `requires_grad` attribute set are updated
  - `functional`: a module(usually imported into the `F` namespace by convention) which contains activation functions, loss functions, etc, as well as non-stateful versions of layers such as convolutional and linear layers.
- `torch.optim`: Contains optimizers such as `SGD`, which update the weights of `Parameter` during the backward step
- `Dataset`: An abstract interface of objects with a `__len__` and a `__getitem__`, including classes provided with Pytorch such as `TensorDataset`
- `DataLoader`: Takes any `Dataset` and creates an iterator which returns batches of data.