Uppsala University

Deep learning

Hand-in assignment (2)

Tong You

May 10, 2021

Contents

1 Introduction **2**

2 PyTorch and MNIST **2**

2.1 Fully-connected networks 2

2.2 Convolutional neural networks 5

3 PyTorch and semantic segmentation **14**

4 Appendix **20**

4.1 PyTorch and MNIST 20

4.2 PyTorch and semantic segmentation 30

1 Introduction

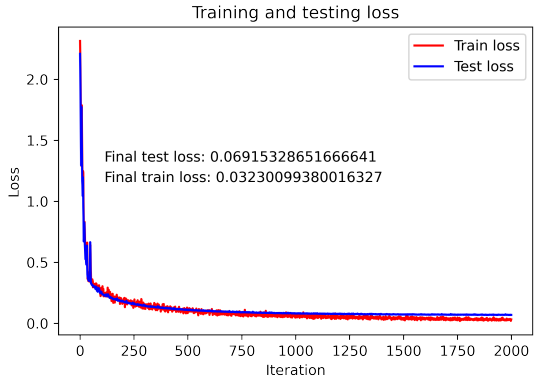
In the previous assignment we had to code a softmax classifier, mini-batched softmax classifier, and a neural network from scratch for the MNIST dataset. For the first part of this assignment we will return to the MNIST dataset, but this time we will be using PyTorch instead. This will also give us CUDA support which will drastically speed-up training. All results will be done on an RTX 2080ti. I will recreate the 2-layer and 4-layer neural networks from assignment 1 in PyTorch instead using hyper parameters that are similar so a one-to-one comparison can be made. We will then explore a convolutional neural network (CNN) for MNIST and try to gain as high accuracy as possible. In the second part of the assignment we will be looking at semantic segmentation on biomedical images, more specifically the Warwick biomedical dataset. For this we will adapt our CNN for MNIST to a fully-convolutional network (FCN). All code is attached to the Appendix.

2 PyTorch and MNIST

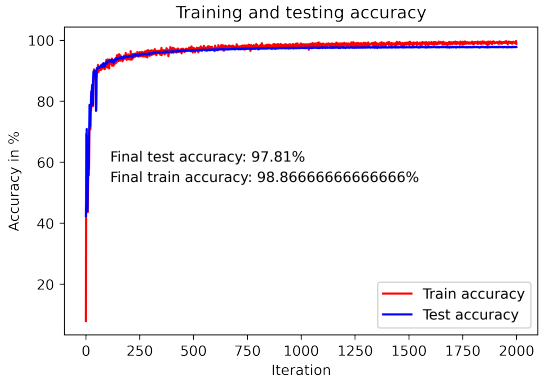
2.1 Fully-connected networks

As mentioned in the previous section, in this section we will be exploring MNIST using PyTorch as our framework. To start with we will re-implement the 2-layer and 4-layer fully-connected networks first. This will allow us to establish a baseline against to compare the CNN and various architectural changes, regularization, and different optimization methods. In the previous assignment we used a mini-batch size of 1500, so we will use this batch-size here as well. We also used quite a high learning rate 0.8 in the previous assignment. This was all with plain SGD and since the 4-layer network design (100 hidden units in all hidden layers) was not optimal, we might expect similar instability in the PyTorch implementation. We will be using ReLU as an activation function for both of these networks. The loss and accuracy plots are shown in Figure 1.

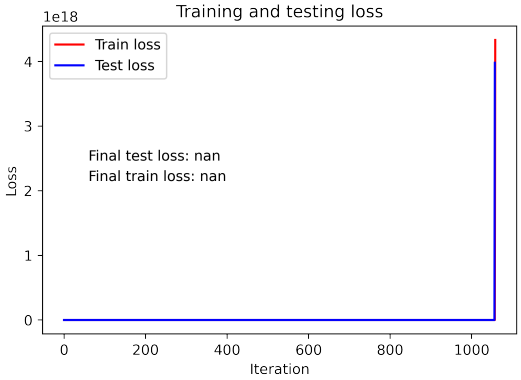
(a) $\gamma = 0.8$ for 2-layer network



(b) $\gamma = 0.8$ for 2-layer network



(c) $\gamma = 0.8$ for 4-layer network



(d) $\gamma = 0.8$ for 4-layer network

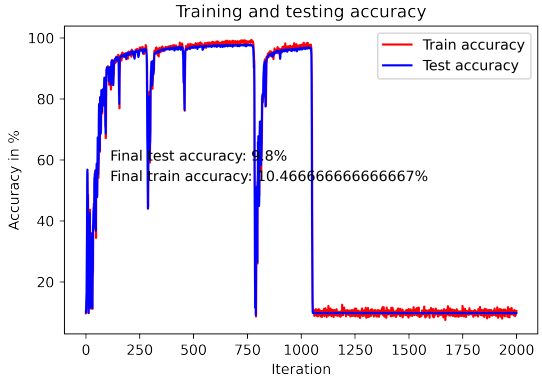
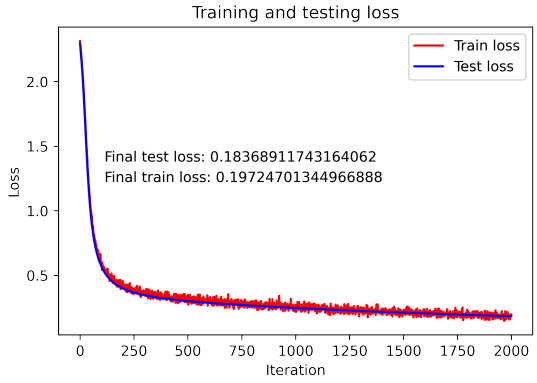


Figure 1: Accuracy and loss plots for 2-layer and 4-layer networks. Note that for these figures we don't plot the accuracy and loss plots every k-th iteration, we plot the value every iteration.

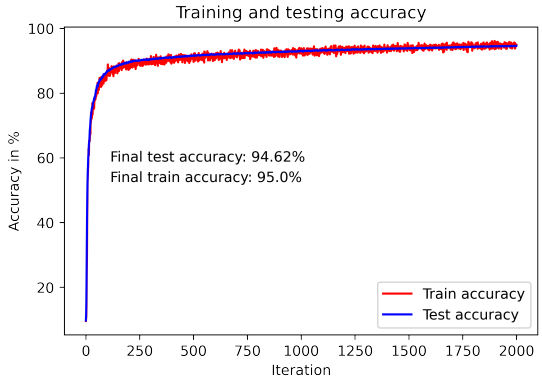
There are several interesting things to observe. We note first of all that the 2-layer network gains a final testing accuracy of 97.81% and the loss and accuracy plots converge in a stable way. For the 4-layer network however, we can see that the network is not stable. In fact the training and testing loss diverge near the end. The large increase in training/testing loss after 1000 iterations corresponds to a corresponding decrease in accuracy to 9.8%. However, it is quite apparent that for the earlier iterations, the accuracy does

reach a value of about 98%. Therefore to investigate whether a lower learning rate would fix such a bug we train both the 2-layer and 4-layer networks with a learning rate of 0.1. The results are shown in Figure 2

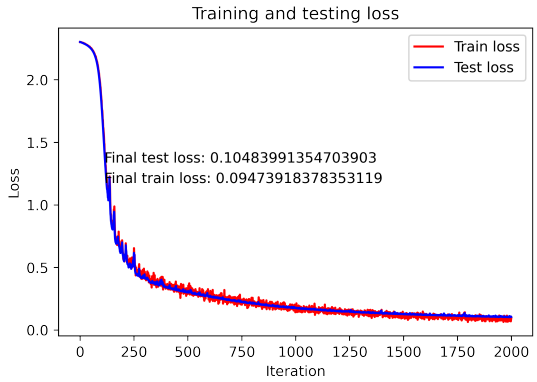
(a) $\gamma = 0.1$ for 2-layer network



(b) $\gamma = 0.1$ for 2-layer network



(c) $\gamma = 0.1$ for 4-layer network



(d) $\gamma = 0.1$ for 4-layer network

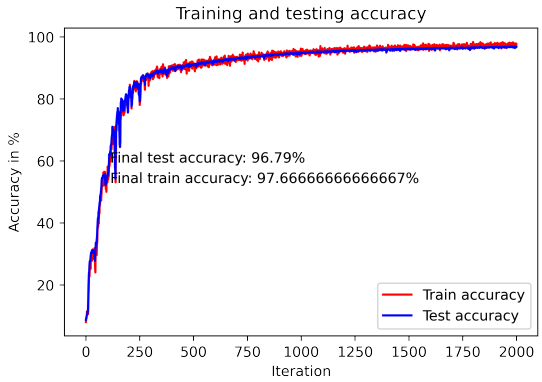


Figure 2: Accuracy and loss plots for 2-layer and 4-layer networks. Note that for these figures we don't plot the accuracy and loss plots every k-th iteration, we plot the value every iteration. The only difference between these plots and the ones in the previous figure are that we change the learning rate to 0.1.

We can see that the 2-layer network again displays stable convergence of the

training/testing loss and accuracies. We do note that the loss and accuracy plots are more stable compared to the 2-layer network with a learning rate of 0.8. The final testing accuracy is also slightly lower at 94.62%. The biggest difference is that the 4-layer network displays much more stable convergence of the loss and accuracies. There is also no huge blow-up anymore. In fact, the 4-layer network converges to a larger final testing accuracy of 96.79% than the 2-layer network. Since the number of hidden units and therefore the width of the networks are the same, we can see that by adding more layers we do get final higher testing accuracy when keeping all other parameters the same. Next we will look how high accuracy we can get by swapping the fully-connected architecture for a convolutional one.

2.2 Convolutional neural networks

We will now explore the MNIST dataset in terms of CNNs. We will start by implementing the following network:

```
conv_mod = nn.Sequential(nn.Conv2d(in_channels = 1,
    out_channels = 8, kernel_size = (3,3), stride = 1, padding
    = 1), nn.ReLU(), nn.MaxPool2d(kernel_size = 2, stride =
    2),
                             nn.Conv2d(in_channels = 8,
                              out_channels = 16, kernel_size =
                              (3,3), stride = 1, padding = 1),
                              nn.ReLU(),
                              nn.MaxPool2d(kernel_size = 2,
                              stride = 2),
                            nn.Conv2d(in_channels = 16,
                             out_channels = 32, kernel_size =
                             (3,3), stride = 1, padding = 1),
                             nn.ReLU(), nn.Flatten(),
                             nn.Linear(32 * 7 * 7, M))
```

We will first train this network using plain SGD and look at the final accuracy. In order to ensure a good trade-off between convergence and waiting too long we will be using a batch-size of 1000. We know that a small batch-size means more iterations per epoch, since we need more passes to sample the entire training set. The opposite holds for larger batch-sizes. It is also well-known

that smaller batch-sizes lead to noisier gradient updates, which also needs to be compensated for by changing the learning rate for example. We end up choosing a learning rate of 0.05. The results are shown in Figure 3.

(a) $\gamma = 0.05$ using SGD (b) $\gamma = 0.05$ using SGD

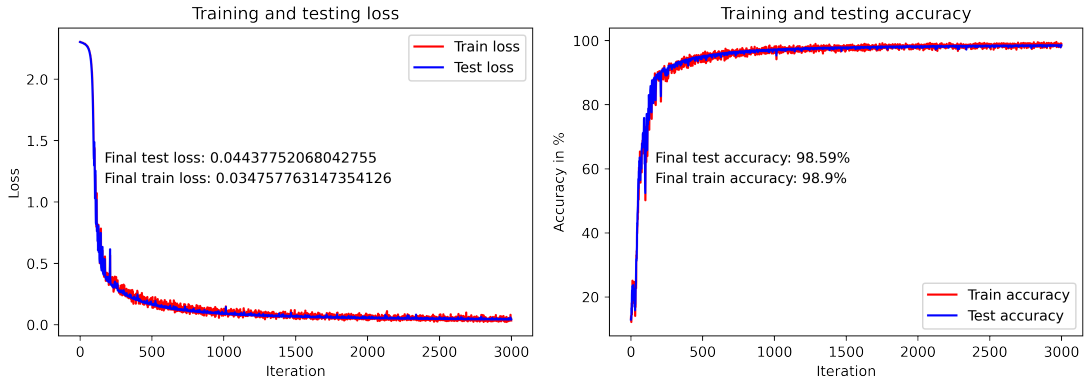


Figure 3: Accuracy and loss plots for CNN. The network is trained with plain SGD.

We can see that the CNN gains a final accuracy of 98.59%. This is higher than the final accuracies reached by the 2-layer and 4-layer networks. Since we take into account spatial relations by not flattening the images into vectors, for a network with optimal settings we should expect to gain higher accuracies. This was all done using plain SGD. What will happen if we choose a different optimizer, while keeping all the other parameters the same. We will investigate this by using ADAM (Adaptive Moment Estimation). For this we will be using the default parameters suggested by PyTorch, and the code looks like:

```
adam_opt = optim.Adam(mod.parameters(), lr = 0.001 , betas =
↪  (0.9, 0.999), eps = 1e-08, weight_decay = 0.0, amsgrad =
↪  False)
```

(a) $\gamma = 0.001$ using ADAM (b) $\gamma = 0.001$ using ADAM

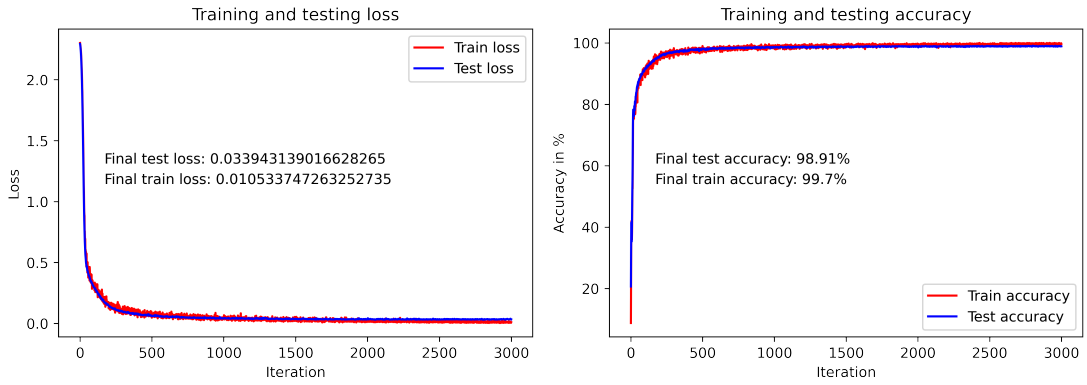


Figure 4: Accuracy and loss plots for CNN. The network is trained with ADAM with default values as suggested by PyTorch.

As we can see, there is a big difference between the CNN trained with plain SGD and the CNN trained with ADAM. For the same batch-size, we can see that training/testing loss and accuracy plots converge in a much more stable way. The final testing accuracy is slightly higher than the CNN optimized with plain SGD: the CNN trained with ADAM gets about 98.91%.

Next we will look at what happens when we swap the order of the activation and pooling layers. Since the pooling layer reduces the dimensionality after application, we will perform a ReLU function on a smaller representation. This should mean that the ReLU should perform faster. Since we don't change anything else, we don't expect a large change in accuracy (except for differences in weight initialisation). The results are shown in Figure 5. Note that we train with SGD since we want to compare to the CNN trained with SGD.

(a) $\gamma = 0.05$ using SGD (b) $\gamma = 0.05$ using SGD

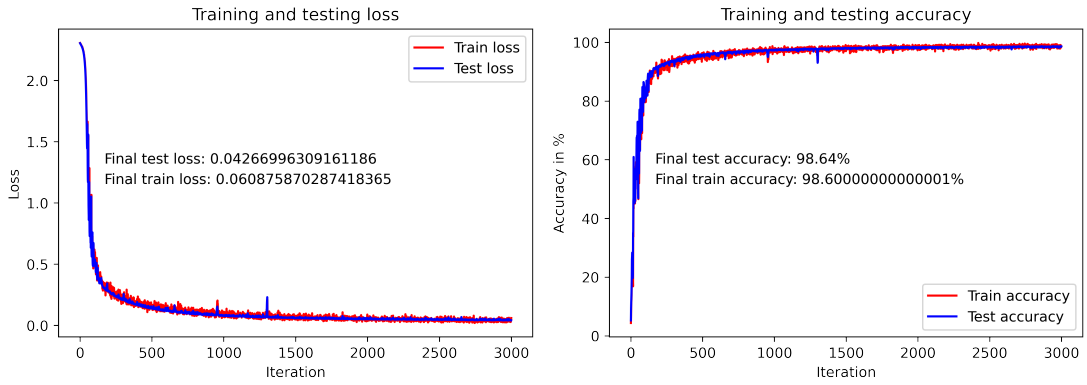


Figure 5: Accuracy and loss plots for CNN. The network is trained with SGD and swapped ReLU and pooling layers.

We can see that compared to Figure 3 the training/testing loss and accuracy plots converge slightly more unstable. However, this could just be due to a difference in initialisation. The final testing accuracy however is quite similar at 98.64%, which is only a 0.05% difference. We will next see whether a swapping of activation and pooling layers is more apparent for a hyperbolic tangent activation function. This is shown in Figure 6.

(a) $\gamma = 0.05$ using SGD (b) $\gamma = 0.05$ using SGD

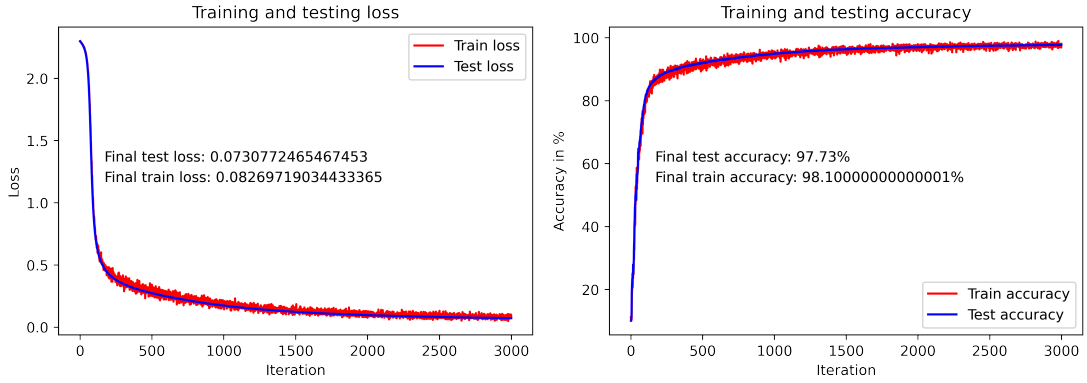


Figure 6: Accuracy and loss plots for CNN. The network is trained with SGD and swapped hyperbolic tangent and pooling layers.

One of the biggest differences between the swapped activation and pooling layers is indeed more visible for hyperbolic tangent activation than ReLU activation. The first thing we note is that the training/testing loss and accuracy plots converge in a more stable way. The biggest difference is that the final testing accuracy is 97.73%. This is a difference of 0.86% which is an order-of-magnitude different. If we look at the PyTorch documentation for the ReLU and hyperbolic tangent activation functions and the equations:

$$ReLU(x) = max(0, x) \tag{1}$$

and

$$Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2}$$

we can see that the Tanh activation saturates at a much smaller value for positive inputs whereas the ReLU activation will increase proportionally for larger positive input values.

Finally, we will look at 3 different ways to try to improve the testing accuracy even more. Since ADAM performs much better than plain SGD, we want to see if there is an improvement when we add batch normalization to the CNN. We will be using a batch-size of 1000 and default ADAM values.

There has been some discussion as to whether we should place the batch normalization layer before or after the non-linearity, in this work we will choose the former. Results are shown in Figure 7.

(a) $\gamma = 0.001$ using ADAM (b) $\gamma = 0.001$ using ADAM

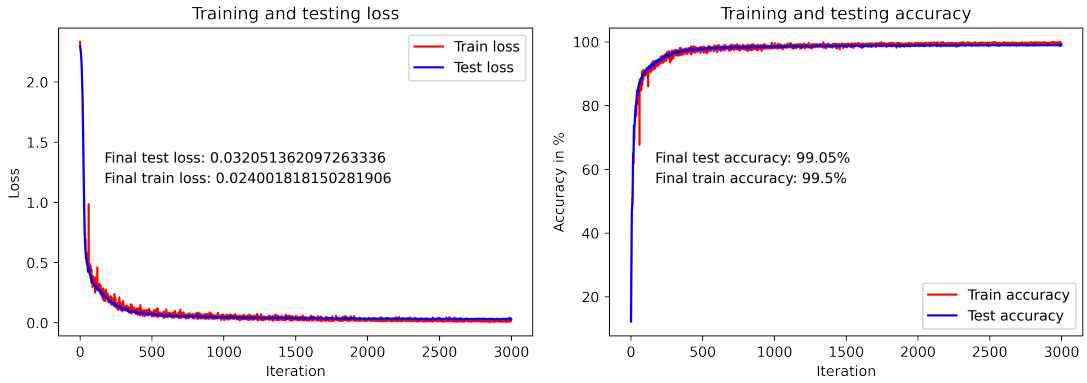


Figure 7: Accuracy and loss plots for CNN. The network is trained with SGD and ReLU with batch normalization before the ReLU.

The final testing accuracy for an ADAM-trained CNN is 99.05%. This is the highest final testing accuracy compared to all previous results. We will explore two more approaches to try to improve the testing accuracy. Next, we will look at adding dropout to the previous model. Following this discussion we will be placing the dropout layer after the ReLU activation and use a dropout probability of 0.5 following the original dropout paper.

(a) $\gamma = 0.001$ using ADAM (b) $\gamma = 0.001$ using ADAM

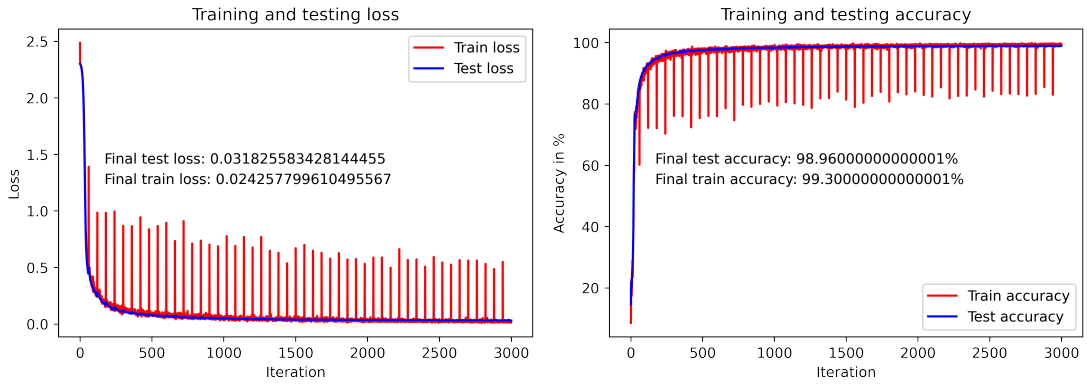


Figure 8: Accuracy and loss plots for CNN. Dropout layers are added after the ReLU activation.

One the biggest difference we can see is that the training loss and accuracy show a very oscillatory behaviour, converging to a final testing accuracy of 98.96%. To test a different type of activation function and see if we can gain back some accuracy, we will use the SELU activation function. We tested the SELU activation both with and without batch normalization and only display the best results. Whereas batch normalization does an explicit normalization to mean zero and unit variance, the SELU activation is supposed to "automatically converge towards zero mean and unit variance" (literal quote).

(a) $\gamma = 0.001$ using ADAM (b) $\gamma = 0.001$ using ADAM

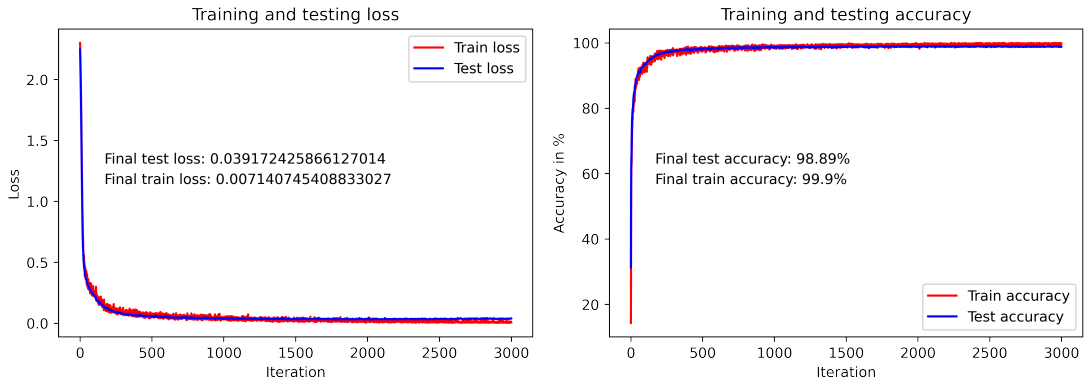


Figure 9: Accuracy and loss plots for CNN. The SELU activation function is used instead of ReLU.

The SELU CNN does not manage to give the best classification results at only a final testing accuracy of 98.89%. The next explorations would be to change the network architecture, but it is interesting to try to optimize the current architecture as much as possible using different optimizers and etc.

To finalize this section, we will be looking at the best-performing model and displaying a confusion matrix which denotes the number of correct classifications on the main diagonal and wrong classifications on the off-diagonals. A wrong classification would be for example an "1" being classified as a "7", or a "6" being classified as an "8" for example. We also show some wrong classifications with the predicted and correct class labels. Since the CNN trained with ADAM gave the best results, we will show the confusion matrix and some wrongly classified images next.

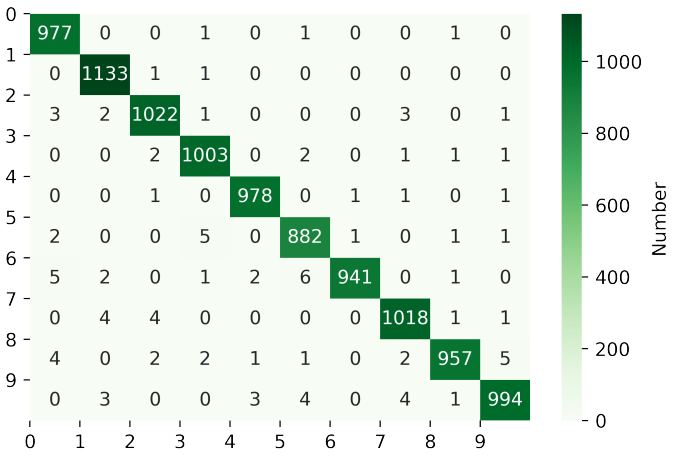


Figure 10: Confusion matrix of best network using ADAM.

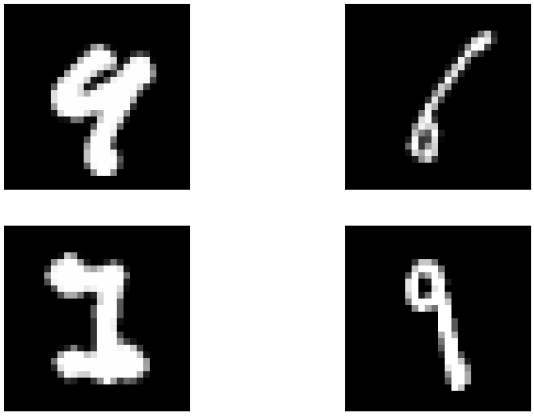


Figure 11: Several wrong classifications using ADAM. From left to right and top to bottom. The correct class is 4 - predicted is 9. The correct class is 6 - predicted is 1. Correct class is 1 - predicted is 3. Correct class is 9 - predicted is 1.

Looking at the distribution of the predictions in the confusion matrix there is not a huge tendency of the network to predict some classes more wrong than others. The example images that were classified wrongly are indeed quite ambiguous so a different approach might yield better results. In the next part we will leave behind MNIST and look at biomedical image segmentation.

3 PyTorch and semantic segmentation

In this section we will explore semantic segmentation of the Warwick biomedical dataset. The dataset contains 80 training images and 60 test images of 128 by 128 pixels. The images provided have the blue channel set to a constant value of zero. In the code attached in the Appendix, this channel is not removed. Our task is a binary one: predict 1 where there is a gland in the testing data and 0 where there is no gland (or background). The provided test labels are binary masks. The training and testing images and segmentation masks are all normalized by dividing by 255. There is an additional step done before we can actually start implementing the training loop. In Pytorch the mini-batches need to have the number of input channels in the second dimensions. Therefore we need to permute the normalized images as follows:

```
xtra_torch = torch.from_numpy(xtra).permute(0, 3, 1,
↪  2).float()
ytra_torch = torch.from_numpy(ytra)
xtes_torch = torch.from_numpy(xtes).permute(0, 3, 1,
↪  2).float()
ytes_torch = torch.from_numpy(ytes)
```

To start with we will start from the CNN architecture for the previous MNIST classification task and adapt it into a fully-connected network. This will be done by replacing the fully-connected layer with a 1x1 convolution with the appropriate number (in our case 2) of output channels. After each pooling layers, our spatial resolution is reduced by 2 (using the same pooling layers as the MNIST CNN). To get an one-to-one correspondence between our CNN output and the segmentation masks we need to upsample the information encoded in the first part of the network. To do this, we will use transposed convolutions. In order to avoid checkerboard artifacts we will use transposed

convolutional layers such that the filter size is an integer multiple of the stride. In what follows, we will use a kernel size of 4x4 and a stride of 2 for each transposed convolution layer. To evaluate the segmentation results on the testing data, we will use the Sørensen–Dice coefficient (DSC for short):

$$DSC(A,B) = \frac{2|A \cap B|}{|A| + |B|} \tag{3}$$

In the above equation A is the segmentation output of the network and B is the provided ground truth segmentation masks. Before we show the final segmentation results and some segmentation outputs we will discuss briefly about the initial network architectures and design choices that were tried. As mentioned earlier, the initial network was an one-to-one copy of the MNIST CNN provided earlier where the fully-connected layer was removed and the rest was kept the same. Using the binary cross entropy loss with logits as our loss function we initially trained with a batch-size of 10 and an ADAM optimizer using the default parameters suggested by PyTorch:

```
adam_opt = optim.Adam(model.parameters(), lr = 0.001, betas =
↪  (0.9, 0.999), eps = 1e-08, weight_decay = 0.0, amsgrad =
↪  False)
```

We also trained for 50 epochs, and after it was observed that the training was quite fast we extended this to 300 epochs to ensure convergence is achieved (important for later). However, the loss didn't decrease and the DSC coefficient stayed constant with spikes at some iterations. Therefore, a different network architecture was used after experimentation with several different candidate networks. This is the final network:

```
segnetbn = nn.Sequential(nn.Conv2d(in_channels = 3,
↪  out_channels = 8, kernel_size = (3,3), stride = 1, padding
↪  = 1), nn.BatchNorm2d(8), nn.ReLU(),
↪  nn.MaxPool2d(kernel_size = 2, stride = 2),
                        nn.Conv2d(in_channels = 8,
                        ↪  out_channels = 16, kernel_size =
                        ↪  (3,3), stride = 1, padding = 1),
                        ↪  nn.BatchNorm2d(16), nn.ReLU(),
                        ↪  nn.MaxPool2d(kernel_size = 2,
                        ↪  stride = 2),
```

```
3                       nn.Conv2d(in_channels = 16,
                        ↪  out_channels = 32, kernel_size =
                        ↪  (3,3), stride = 1, padding = 1),
                        ↪  nn.BatchNorm2d(32), nn.ReLU(),
                        ↪  nn.MaxPool2d(kernel_size = 2,
                        ↪  stride = 2),
4                       nn.Conv2d(in_channels = 32,
                        ↪  out_channels = 64, kernel_size =
                        ↪  (3,3), stride = 1, padding = 1),
                        ↪  nn.BatchNorm2d(64), nn.ReLU(),
5                       nn.ConvTranspose2d(in_channels = 64,
                        ↪  out_channels = 32, kernel_size =
                        ↪  (4,4), stride = 2, padding = 1),
6                       nn.ConvTranspose2d(in_channels = 32,
                        ↪  out_channels = 16, kernel_size =
                        ↪  (4,4), stride = 2, padding = 1),
7                       nn.ConvTranspose2d(in_channels = 16,
                        ↪  out_channels = 8, kernel_size =
                        ↪  (4,4), stride = 2, padding = 1),
8                       nn.Conv2d(in_channels = 8,
                        ↪  out_channels = 2, kernel_size =
                        ↪  (1,1), stride = 1, padding = 0),
                        ↪  nn.Sigmoid())
```

Importantly, batch-normalization layers were added before each ReLU non-linearity and an additional convolutional layer was added to increase the number of channels to 64. This network was trained with ADAM for 300 epochs with default parameters (as mentioned previously). The mini-batch size was 10 and evaluation was done on the entire test data set. A final curiosity about the network is that despite the fact that the used loss function actually combines a sigmoid layer and a binary cross entropy loss function the network still outputs negative predictions. We reduced the loss by adding a sigmoid function to the final layer. Since we know that the sigmoid outputs values between 0 and 1, and our segmentation masks are also between 0 and 1 the network will most likely converge in a more stable way. Results are shown in Figure 12.

(a) Binary cross entropy loss (b) Dice coefficient

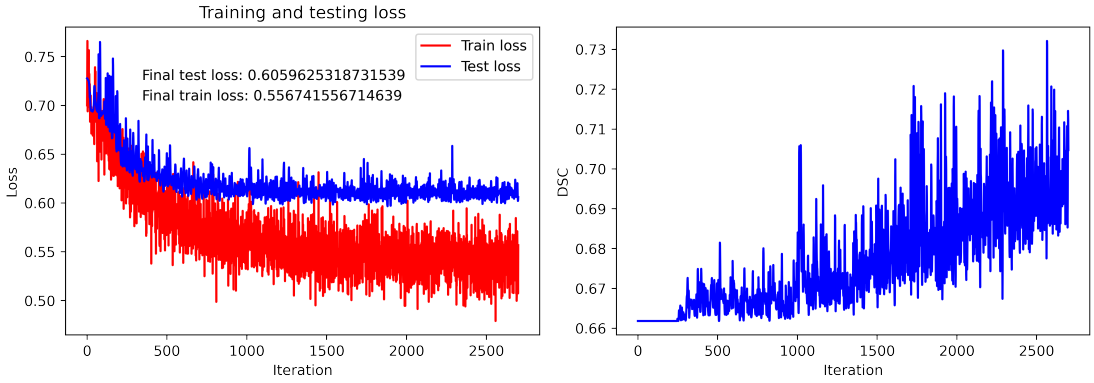


Figure 12: Training and testing loss of the segmentation network on the left. The testing loss plateaus at a larger value than the training loss. There is no indication of overfitting. On the right, the dice coefficient evaluated on the entire testing data is shown. The final dice coefficient is about 0.7047.

The final training and testing losses are indicated in the above caption, as well as the final dice coefficient. We can see that the dice coefficient does increase over time, however the increase is slow. Next we will show one example of a prediction on the training data with the correct mask.

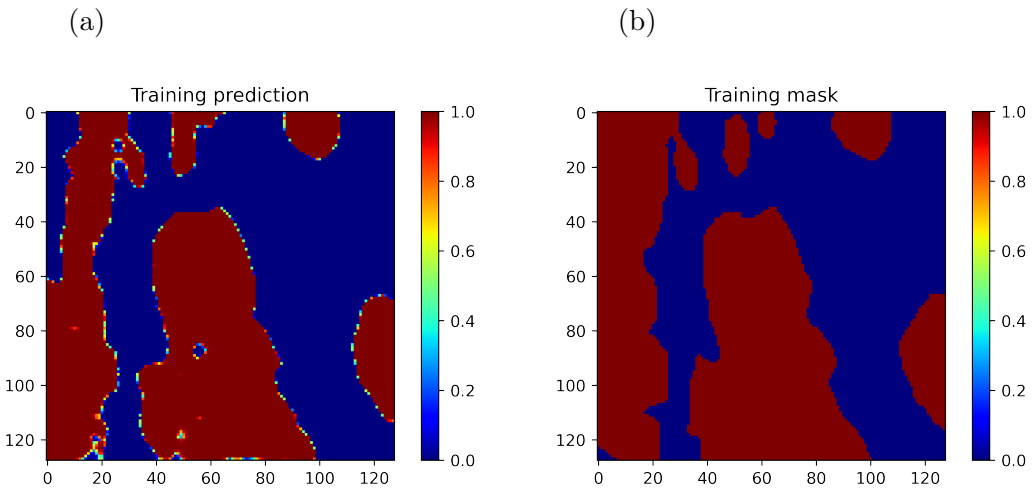
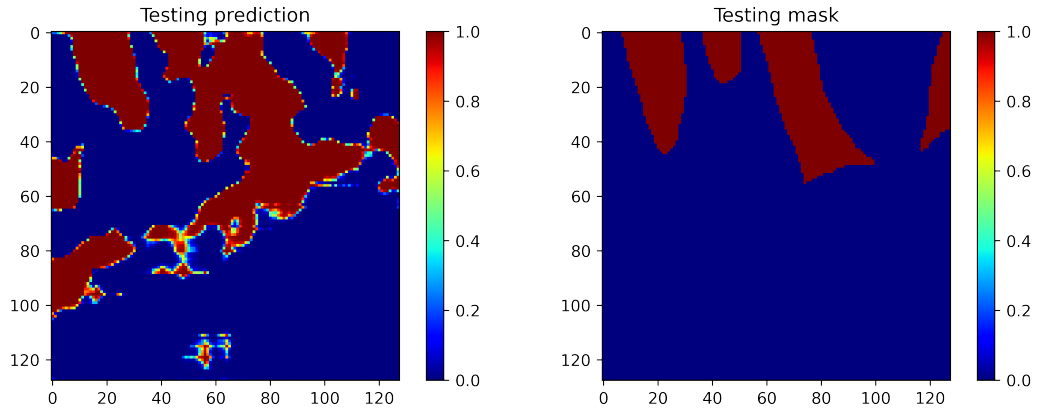


Figure 13: An example from the first of the 10 final mini-batches with the network prediction on the left and the ground truth mask on the right. The color map was chosen such that the differences between prediction and ground truth mask can be observed clearly.

We finally look at two examples out of the sixty testing images with network prediction and ground truth masks as a comparison. Since the testing data was not shuffled the two chosen test images correspond directly to the provided data set.

(a) Test image 10

(b) Test mask 10



(c) Test image 20

(d) Test image 20

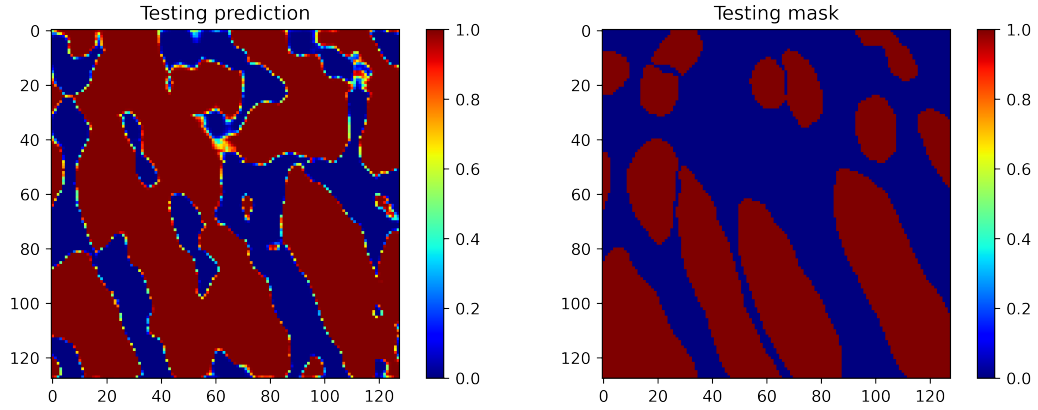


Figure 14: Network predictions and ground truth masks of the testing data. All network predictions are shown on the left and all masks on the right. The number of the image is indicated above the plot.

In the above figure we can see that the network predictions for both image 10 and image 20 are far off from the ground truth mask. In image 10 we can see in fact that the network predicted glands in locations where is not even a gland in the ground truth. This is less apparent for image 20 since the

ground truth is not as sparse as image 10. We suspect this has something to do with the fact that the convolutional kernels have too small kernel sizes. So to aggregate more information about the actual image it might be beneficial to try to increase the kernel size at the same time as we increase the channel depth. However, it is promising that at locations where there is actually a gland the network managed to predict a gland as well. It might be more harmful to miss glands than to wrongfully predict a gland, however it is obviously better to not predict a gland where there is none. The examples chosen represent a dense and sparse image with varying shapes. Test images that contained many similar objects seemed to be predicted correctly more often since the network can re-use the information it has encoded for one object. This is an advantage of the convolutional approach.

4 Appendix

All models are implemented in Jupyter notebooks to allow for easy experimentation. We will attach the code in this report as a single file. The code will also be separately attached as two separate Jupyter notebooks for the MNIST and semantic segmentation task.

4.1 PyTorch and MNIST

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

from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader

import numpy as np
import seaborn as sns

from matplotlib import pyplot as plt
from sklearn.metrics import confusion_matrix

from load_mnist import load_mnist
```

```
16
17 # Loading training and testing data
18 xtrain, ytrain, xtest, ytest = load_mnist()
19
20 # Reset function
21 def reset_model(model):
22     for layer in model.children():
23         if hasattr(layer, 'reset_parameters'):
24             layer.reset_parameters()
25
26 # Get dimensions of training and testing data
27 M = ytrain.shape[1]
28 p = xtrain.shape[1]
29
30 ntrain = xtrain.shape[0]
31 ntest = xtest.shape[0]
32
33 # Converting training and testing data to torch tensors
34 xtra_torch = torch.tensor(xtrain).float()
35 ytra_torch = torch.tensor(ytrain).float()
36 xtes_torch = torch.tensor(xtest).float()
37 ytes_torch = torch.tensor(ytest).float()
38
39 # Converting our flat vectors to images for the CNNs
40 xtra_conv = torch.tensor(xtrain).reshape(ntrain, 28,
   ↪  28).float().unsqueeze(1)
41 ytra_conv = torch.tensor(ytrain).float()
42 xtes_conv = torch.tensor(xtest).reshape(ntest, 28,
   ↪  28).float().unsqueeze(1)
43 ytes_conv = torch.tensor(ytest).float()
44
45 # Check if CUDA is available
46 device = torch.device("cuda") if torch.cuda.is_available()
   ↪  else torch.device("cpu")
47 print("Device:", torch.cuda.get_device_name(device))
48
49 # FCN
50 num_batch = 1000
```

```
training_data = TensorDataset(xtra_torch.to(device),
↪  ytra_torch.to(device))
train_dat_fcn = DataLoader(training_data, shuffle = True,
↪  batch_size = num_batch)

xte = xtes_torch.to(device)
yte = ytes_torch.to(device)

# CNN
train_dat_im = TensorDataset(xtra_conv.to(device),
↪  ytra_conv.to(device))
train_dat_cnn = DataLoader(train_dat_im, shuffle = True,
↪  batch_size = num_batch)

xte_im = xtes_conv.to(device)
yte_im = ytes_conv.to(device)

# 2-layer network
model_2L = nn.Sequential(nn.Linear(p, 100), nn.ReLU(),
↪  nn.Linear(100, M))

# 4-layer network
model_4L = nn.Sequential(nn.Linear(p, 256), nn.ReLU(),
↪  nn.Linear(256, 128), nn.ReLU(), nn.Linear(128, 64),
↪  nn.ReLU(), nn.Linear(64, M))

# Convolutional network
conv_mod = nn.Sequential(nn.Conv2d(in_channels = 1,
↪  out_channels = 8, kernel_size = (3,3), stride = 1, padding
↪  = 1), nn.ReLU(), nn.MaxPool2d(kernel_size = 2, stride =
↪  2),
                         nn.Conv2d(in_channels = 8,
                         ↪  out_channels = 16, kernel_size =
                         ↪  (3,3), stride = 1, padding = 1),
                         ↪  nn.ReLU(),
                         ↪  nn.MaxPool2d(kernel_size = 2,
                         ↪  stride = 2),
```

```
73                              nn.Conv2d(in_channels = 16,
                                ↪  out_channels = 32, kernel_size =
                                ↪  (3,3), stride = 1, padding = 1),
                                ↪  nn.ReLU(), nn.Flatten(),
                                ↪  nn.Linear(32 * 7 * 7, M))
74
75  conv_swap = nn.Sequential(nn.Conv2d(in_channels = 1,
    ↪  out_channels = 8, kernel_size = (3,3), stride = 1, padding
    ↪  = 1), nn.MaxPool2d(kernel_size = 2, stride = 2),
    ↪  nn.ReLU(),
76                              nn.Conv2d(in_channels = 8,
                                ↪  out_channels = 16, kernel_size =
                                ↪  (3,3), stride = 1, padding = 1),
                                ↪  nn.MaxPool2d(kernel_size = 2,
                                ↪  stride = 2), nn.ReLU(),
77                             nn.Conv2d(in_channels = 16,
                               ↪  out_channels = 32, kernel_size =
                               ↪  (3,3), stride = 1, padding = 1),
                               ↪  nn.ReLU(), nn.Flatten(),
                               ↪  nn.Linear(32 * 7 * 7, M))
78
79  conv_swap_htan = nn.Sequential(nn.Conv2d(in_channels = 1,
    ↪  out_channels = 8, kernel_size = (3,3), stride = 1, padding
    ↪  = 1), nn.MaxPool2d(kernel_size = 2, stride = 2),
    ↪  nn.Tanh(),
80                              nn.Conv2d(in_channels = 8,
                                ↪  out_channels = 16, kernel_size =
                                ↪  (3,3), stride = 1, padding = 1),
                                ↪  nn.MaxPool2d(kernel_size = 2,
                                ↪  stride = 2), nn.Tanh(),
81                             nn.Conv2d(in_channels = 16,
                               ↪  out_channels = 32, kernel_size =
                               ↪  (3,3), stride = 1, padding = 1),
                               ↪  nn.Tanh(), nn.Flatten(),
                               ↪  nn.Linear(32 * 7 * 7, M))
82
```

```
conv_mod_bn = nn.Sequential(nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = (3,3), stride = 1, padding = 1), nn.BatchNorm2d(8), nn.ReLU(), nn.MaxPool2d(kernel_size = 2, stride = 2),
                            nn.Conv2d(in_channels = 8, out_channels = 16, kernel_size = (3,3), stride = 1, padding = 1), nn.BatchNorm2d(16), nn.ReLU(), nn.MaxPool2d(kernel_size = 2, stride = 2),
                           nn.Conv2d(in_channels = 16, out_channels = 32, kernel_size = (3,3), stride = 1, padding = 1), nn.BatchNorm2d(32), nn.ReLU(), nn.Flatten(), nn.Linear(32 * 7 * 7, M))

conv_mod_bn_dropout = nn.Sequential(nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = (3,3), stride = 1, padding = 1), nn.BatchNorm2d(8), nn.ReLU(), nn.Dropout2d(0.5), nn.MaxPool2d(kernel_size = 2, stride = 2),
                            nn.Conv2d(in_channels = 8, out_channels = 16, kernel_size = (3,3), stride = 1, padding = 1), nn.BatchNorm2d(16), nn.ReLU(), nn.Dropout2d(0.5), nn.MaxPool2d(kernel_size = 2, stride = 2),
                           nn.Conv2d(in_channels = 16, out_channels = 32, kernel_size = (3,3), stride = 1, padding = 1), nn.BatchNorm2d(32), nn.ReLU(), nn.Dropout2d(0.5), nn.Flatten(), nn.Linear(32 * 7 * 7, M))

```

```
conv_mod_bn_selu = nn.Sequential(nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = (3,3), stride = 1, padding = 1), nn.SELU(), nn.MaxPool2d(kernel_size = 2, stride = 2),
                            nn.Conv2d(in_channels = 8, out_channels = 16, kernel_size = (3,3), stride = 1, padding = 1), nn.SELU(), nn.MaxPool2d(kernel_size = 2, stride = 2),
                           nn.Conv2d(in_channels = 16, out_channels = 32, kernel_size = (3,3), stride = 1, padding = 1), nn.SELU(), nn.Flatten(), nn.Linear(32 * 7 * 7, M))


# Selecting model
mod = conv_mod

# Resetting model when we switch architectures
reset_model(mod)

# Moving model to GPU
mod.to(device)

# Check dimensions through network
check_dims = False
if check_dims == True:
    X = torch.rand(size = (10, 1, 28, 28)).to(device)
    for layer in mod:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

# Loss function
loss = F.cross_entropy
it = 0
```

```
116 loss_train , acc_train = [], []
117 loss_test, acc_test = [], []
118 l_rate = 0.05
119
120 # Vanilla SGD optimizer
121 sgd_opt = optim.SGD(mod.parameters(), lr = l_rate)
122
123 # Adam optimizer
124 adam_opt = optim.Adam(mod.parameters(), lr = 0.001 , betas =
    ↪  (0.9, 0.999), eps = 1e-08, weight_decay = 0, amsgrad =
    ↪  False)
125
126 # Exercise 1.5 - choosing three different optimizers
127 # Adadelta optimizer
128 adad_opt = optim.Adadelta(mod.parameters(), lr = 1.0, rho =
    ↪  0.9, eps = 1e-06, weight_decay = 0)
129
130 # Adagrad optimizer
131 adag_opt = optim.Adagrad(mod.parameters(), lr = 0.01, lr_decay
    ↪  = 0, weight_decay = 0, initial_accumulator_value = 0, eps
    ↪  = 1e-10)
132
133 # Adamax optimizer
134 adamax_opt = optim.Adamax(mod.parameters(), lr = 0.002, betas
    ↪  = (0.9, 0.999), eps = 1e-08, weight_decay = 0)
135
136 # ASGD optimizer
137 asgd_opt = optim.ASGD(mod.parameters(), lr = 0.01, lambd =
    ↪  0.0001, alpha = 0.75, t0 = 1000000.0, weight_decay = 0)
138
139 # RMSprop optimizer
140 rmsp_opt = optim.RMSprop(mod.parameters(), lr = 0.01, alpha =
    ↪  0.99, eps = 1e-08, weight_decay = 0, momentum = 0,
    ↪  centered = False)
141
142 # Selecting optimizer
143 optr = adam_opt
144 full_n = True
```

```
145
146 if full_n == False:
147     training_dat = train_dat_fcn
148     xt = xte
149     yt = yte
150 else:
151     training_dat = train_dat_cnn
152     xt = xte_im
153     yt = yte_im
154
155 # Optimization loop
156 num_epochs = 50
157
158 y_predictions = []
159
160 for epoch in range(num_epochs):
161     mod.train()
162     for xbatch, ybatch in training_dat:
163         optr.zero_grad()
164         prediction = mod(xbatch)
165         pred = torch.argmax(prediction, dim = 1)
166
167         true_lab = torch.argmax(ybatch, dim = 1)
168
169         ce_loss = loss(prediction, true_lab)
170
171         acc_train.append(100 * (1/num_batch) * torch.sum(pred
            ↪  == true_lab, dim = 0).item())
172
173         loss_train.append(ce_loss.item())
174
175         ce_loss.backward()
176         optr.step()
177
178         mod.eval()
179         with torch.no_grad():
180             ytrue_lab = torch.argmax(yt, dim = 1)
181             prediction_test = mod(xt)
```

```
182             pred_test = torch.argmax(prediction_test, dim = 1)
183
184             ce_test = loss(prediction_test, ytrue_lab)
185             loss_test.append(ce_test.item())
186             acc_test.append(100 * (1/ntest) *
                ↪  torch.sum(pred_test == ytrue_lab, dim =
                ↪  0).item())
187
188             # Saving final predictions
189             if epoch == (num_epochs - 1):
190                 y_predictions = mod(xt)
191
192         it += 1
193     print("Epoch %s/%s" % (epoch + 1, num_epochs))
194
195 plt.figure(1)
196 loss_tra, = plt.plot(loss_train, 'r')
197 loss_tes, = plt.plot(loss_test, 'b')
198 plt.title("Training and testing loss")
199 plt.xlabel("Iteration")
200 plt.ylabel("Loss")
201 plt.legend([loss_tra, loss_tes], ['Train loss', 'Test loss'])
202 plt.annotate("Final train loss: %s" % (loss_train[-1])
    ↪  ,xycoords = 'figure fraction', xy = (0.2,0.5))
203 plt.annotate("Final test loss: %s" % (loss_test[-1]), xycoords
    ↪  = 'figure fraction', xy = (0.2,0.55))
204 print("Final training loss: %s." % loss_train[-1])
205 print("Final testing loss: %s." % loss_test[-1])
206 plt.savefig("loss_plots", dpi = 500)
207
208 plt.figure(2)
209 acc_tra, = plt.plot(acc_train, 'r')
210 acc_tes, = plt.plot(acc_test, 'b')
211 plt.title("Training and testing accuracy")
212 plt.xlabel("Iteration")
213 plt.ylabel("Accuracy in %")
214 plt.legend([acc_tra, acc_tes], ['Train accuracy', 'Test
    ↪  accuracy'])
```

```
plt.annotate("Final train accuracy: %s%%" % (acc_train[-1])
 ↪  ,xycoords = 'figure fraction', xy = (0.2,0.5))
plt.annotate("Final test accuracy: %s%%" % (acc_test[-1]),
 ↪  xycoords = 'figure fraction', xy = (0.2,0.55))
print()
print("Final training accuracy: %s%%." % acc_train[-1])
print("Final testing accuracy: %s%%." % acc_test[-1])
plt.savefig("acc_plots", dpi = 500)

# Retrieve data and labels from the GPU
y_test = yte_im.data.cpu()

# Remove singleton dimension
y_test = np.squeeze(y_test, axis = 1)
ytrue = torch.argmax(y_test, dim = 1)

ypreds = y_predictions.data.cpu()
ypredslab = torch.argmax(ypreds, dim = 1)

cf_mat = confusion_matrix(ytrue, ypredslab, normalize = None)

mnist_classes = np.arange(10)
mnist_lab = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

if True:
    plt.figure(3)
    cmat = sns.heatmap(cf_mat, cmap = 'Greens', annot = True,
     ↪  fmt = "d", cbar_kws = {'label':'Number'}) # use .3f
     ↪  for floating-point data
    cmat.set_xticklabels(mnist_lab, rotation = 'horizontal')
    cmat.set_yticklabels(mnist_lab, rotation = 'horizontal')
    plt.xticks(mnist_classes, mnist_lab)
    plt.yticks(mnist_classes, mnist_lab)
    plt.savefig("confusion_mat", dpi = 500)

# Showing 3 examples of wrong classifications
wrong_examples = (ytrue != ypredslab)

```

```
wrong_preds = np.squeeze(xtes_conv[wrong_examples])
wrong_labs = ypredslab[wrong_examples]
actual_labs = torch.argmax(ytes_conv[wrong_examples], dim = 1)

ind = np.random.choice(a = wrong_labs.shape[0], size = 4,
↪  replace = True)

fig, ax = plt.subplots(2 , 2)
for a in fig.axes:
    a.set_xticks([])
    a.set_yticks([])

ax[0,0].imshow(wrong_preds[ind[0], :, :], cmap = 'gray')
ax[0,1].imshow(wrong_preds[ind[1], :, :], cmap = 'gray')
ax[1,0].imshow(wrong_preds[ind[2], :, :], cmap = 'gray')
ax[1,1].imshow(wrong_preds[ind[3], :, :], cmap = 'gray')
plt.savefig("wrong_preds", dpi = 500)

for i in ind:
    print("Correct label: %d" % actual_labs[i].item())
    print("Predicted label: %d" % wrong_labs[i].item())
```

4.2 PyTorch and semantic segmentation

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

import seaborn as sn
import numpy as np
from sklearn.metrics import confusion_matrix

from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader

from load_warwick import load_warwick
```

```
14
15 from matplotlib import pyplot as plt
16
17 # Loading data
18 # Training data: 85 images of 128x128x3
19 # Training labels: 85 images of 128x128
20 # Testing data: 60 images of 128x128x3
21 # Testing labels: 60 images of 128x128
22 xtra, ytra, xtes, ytes = load_warwick()
23
24 # Resetting model function
25 # Credits:
   ↪ https://discuss.pytorch.org/t/reset-model-weights/19180/4
26 def reset_model(model):
27     for layer in model.children():
28        if hasattr(layer, 'reset_parameters'):
29            layer.reset_parameters()
30
31 # Sorensen-Dice coefficient
32 def dsc(A, B):
33
34     a = A.bool()
35     b = B.bool()
36
37     intersect = torch.logical_and(a,b)
38
39     dice_coeff = (2.0 * intersect.sum()) / (a.sum() + b.sum())
40
41     return dice_coeff
42
43 # Preprocessing of train/test data by dividing with maximum
   ↪ intensity
44 xtra = xtra / 255
45 ytra = ytra / 255
46
47 xtes = xtes / 255
48 ytes = ytes / 255
49
```

```
# Get some dimensions
pixels = 128
ntrain = xtra.shape[0]
ntest = xtes.shape[0]

# Convert to Torch tensors
xtra_torch = torch.from_numpy(xtra).permute(0, 3, 1, 2).float()
ytra_torch = torch.from_numpy(ytra)
xtes_torch = torch.from_numpy(xtes).permute(0, 3, 1, 2).float()
ytes_torch = torch.from_numpy(ytes)

# Check if CUDA is available
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", torch.cuda.get_device_name(device))

# Creating data loader for training data
num_batch = 10
training_set = TensorDataset(xtra_torch.to(device), ytra_torch.to(device))
training_loader = DataLoader(training_set, shuffle = True, batch_size = num_batch)

xtest = xtes_torch.to(device)
ytest = ytes_torch.to(device)

# Convolutional network
segnet = nn.Sequential(nn.Conv2d(in_channels = 3, out_channels = 8, kernel_size = (3,3), stride = 1, padding = 1), nn.ReLU(), nn.MaxPool2d(kernel_size = 2, stride = 2),
                       nn.Conv2d(in_channels = 8, out_channels = 16, kernel_size = (3,3), stride = 1, padding = 1), nn.ReLU(), nn.MaxPool2d(kernel_size = 2, stride = 2),
```

```
76                          nn.Conv2d(in_channels = 16,
                            ↪  out_channels = 32, kernel_size =
                            ↪  (3,3), stride = 1, padding = 1),
                            ↪  nn.ReLU(),
                            ↪  nn.MaxPool2d(kernel_size = 2,
                            ↪  stride = 2),
77                          nn.Conv2d(in_channels = 32,
                            ↪  out_channels = 64, kernel_size =
                            ↪  (3,3), stride = 1, padding = 1),
                            ↪  nn.ReLU(),
78                          nn.ConvTranspose2d(in_channels = 64,
                            ↪  out_channels = 32, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
79                          nn.ConvTranspose2d(in_channels = 32,
                            ↪  out_channels = 16, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
80                          nn.ConvTranspose2d(in_channels = 16,
                            ↪  out_channels = 8, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
81                          nn.Conv2d(in_channels = 8,
                            ↪  out_channels = 2, kernel_size =
                            ↪  (1,1), stride = 1, padding = 0))
82
83  segnetbn = nn.Sequential(nn.Conv2d(in_channels = 3,
    ↪  out_channels = 8, kernel_size = (3,3), stride = 1, padding
    ↪  = 1), nn.BatchNorm2d(8), nn.ReLU(),
    ↪  nn.MaxPool2d(kernel_size = 2, stride = 2),
84                          nn.Conv2d(in_channels = 8,
                            ↪  out_channels = 16, kernel_size =
                            ↪  (3,3), stride = 1, padding = 1),
                            ↪  nn.BatchNorm2d(16), nn.ReLU(),
                            ↪  nn.MaxPool2d(kernel_size = 2,
                            ↪  stride = 2),
```

```
85                          nn.Conv2d(in_channels = 16,
                            ↪  out_channels = 32, kernel_size =
                            ↪  (3,3), stride = 1, padding = 1),
                            ↪  nn.BatchNorm2d(32), nn.ReLU(),
                            ↪  nn.MaxPool2d(kernel_size = 2,
                            ↪  stride = 2),
86                          nn.Conv2d(in_channels = 32,
                            ↪  out_channels = 64, kernel_size =
                            ↪  (3,3), stride = 1, padding = 1),
                            ↪  nn.BatchNorm2d(64), nn.ReLU(),
87                          nn.ConvTranspose2d(in_channels = 64,
                            ↪  out_channels = 32, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
88                          nn.ConvTranspose2d(in_channels = 32,
                            ↪  out_channels = 16, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
89                          nn.ConvTranspose2d(in_channels = 16,
                            ↪  out_channels = 8, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
90                          nn.Conv2d(in_channels = 8,
                            ↪  out_channels = 2, kernel_size =
                            ↪  (1,1), stride = 1, padding = 0),
                            ↪  nn.Sigmoid())
91
92  m_seg_selu = nn.Sequential(nn.Conv2d(in_channels = 3,
    ↪  out_channels = 8, kernel_size = (3,3), stride = 1, padding
    ↪  = 1), nn.SELU(), nn.MaxPool2d(kernel_size = 2, stride =
    ↪  2),
93                          nn.Conv2d(in_channels = 8,
                            ↪  out_channels = 16, kernel_size =
                            ↪  (3,3), stride = 1, padding = 1),
                            ↪  nn.SELU(),
                            ↪  nn.MaxPool2d(kernel_size = 2,
                            ↪  stride = 2),
```

```
94                          nn.Conv2d(in_channels = 16,
                            ↪  out_channels = 32, kernel_size =
                            ↪  (3,3), stride = 1, padding = 1),
                            ↪  nn.SELU(),
                            ↪  nn.MaxPool2d(kernel_size = 2,
                            ↪  stride = 2),
95                          nn.Conv2d(in_channels = 32,
                            ↪  out_channels = 64, kernel_size =
                            ↪  (3,3), stride = 1, padding = 1),
                            ↪  nn.SELU(),
96                          nn.ConvTranspose2d(in_channels = 64,
                            ↪  out_channels = 32, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
97                          nn.ConvTranspose2d(in_channels = 32,
                            ↪  out_channels = 16, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
98                          nn.ConvTranspose2d(in_channels = 16,
                            ↪  out_channels = 8, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
99                          nn.Conv2d(in_channels = 8,
                            ↪  out_channels = 2, kernel_size =
                            ↪  (1,1), stride = 1, padding = 0))
100
101 m_seg_k_big = nn.Sequential(nn.Conv2d(in_channels = 3,
    ↪  out_channels = 8, kernel_size = (5,5), stride = 1, padding
    ↪  = 2), nn.BatchNorm2d(8), nn.ReLU(),
    ↪  nn.MaxPool2d(kernel_size = 2, stride = 2),
102                         nn.Conv2d(in_channels = 8,
                            ↪  out_channels = 16, kernel_size =
                            ↪  (5,5), stride = 1, padding = 2),
                            ↪  nn.BatchNorm2d(16), nn.ReLU(),
                            ↪  nn.MaxPool2d(kernel_size = 2,
                            ↪  stride = 2),
```

```
103                         nn.Conv2d(in_channels = 16,
                            ↪  out_channels = 32, kernel_size =
                            ↪  (5,5), stride = 1, padding = 2),
                            ↪  nn.BatchNorm2d(32), nn.ReLU(),
                            ↪  nn.MaxPool2d(kernel_size = 2,
                            ↪  stride = 2),
104                         nn.Conv2d(in_channels = 32,
                            ↪  out_channels = 64, kernel_size =
                            ↪  (5,5), stride = 1, padding = 2),
                            ↪  nn.BatchNorm2d(64), nn.ReLU(),
                            ↪  nn.MaxPool2d(kernel_size = 2,
                            ↪  stride = 2),
105                         nn.Conv2d(in_channels = 64,
                            ↪  out_channels = 128, kernel_size =
                            ↪  (5,5), stride = 1, padding = 2),
                            ↪  nn.BatchNorm2d(128), nn.ReLU(),
106                         nn.ConvTranspose2d(in_channels = 128,
                            ↪  out_channels = 64, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
107                         nn.ConvTranspose2d(in_channels = 64,
                            ↪  out_channels = 32, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
108                         nn.ConvTranspose2d(in_channels = 32,
                            ↪  out_channels = 16, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
109                         nn.ConvTranspose2d(in_channels = 16,
                            ↪  out_channels = 8, kernel_size =
                            ↪  (4,4), stride = 2, padding = 1),
110                         nn.Conv2d(in_channels = 8,
                            ↪  out_channels = 2, kernel_size =
                            ↪  (1,1), stride = 1, padding = 0))
111
112
113 m_seg_k_big_selu = nn.Sequential(nn.Conv2d(in_channels = 3,
    ↪  out_channels = 8, kernel_size = (5,5), stride = 1, padding
    ↪  = 2), nn.SELU(), nn.MaxPool2d(kernel_size = 2, stride =
    ↪  2),
```

```
114                     nn.Conv2d(in_channels = 8,
                        ↪  out_channels = 16, kernel_size =
                        ↪  (5,5), stride = 1, padding = 2),
                        ↪  nn.SELU(),
                        ↪  nn.MaxPool2d(kernel_size = 2,
                        ↪  stride = 2),
115                     nn.Conv2d(in_channels = 16,
                        ↪  out_channels = 32, kernel_size =
                        ↪  (5,5), stride = 1, padding = 2),
                        ↪  nn.SELU(),
                        ↪  nn.MaxPool2d(kernel_size = 2,
                        ↪  stride = 2),
116                     nn.Conv2d(in_channels = 32,
                        ↪  out_channels = 64, kernel_size =
                        ↪  (5,5), stride = 1, padding = 2),
                        ↪  nn.SELU(),
                        ↪  nn.MaxPool2d(kernel_size = 2,
                        ↪  stride = 2),
117                     nn.Conv2d(in_channels = 64,
                        ↪  out_channels = 128, kernel_size =
                        ↪  (5,5), stride = 1, padding = 2),
                        ↪  nn.SELU(),
118                     nn.ConvTranspose2d(in_channels = 128,
                        ↪  out_channels = 64, kernel_size =
                        ↪  (4,4), stride = 2, padding = 1),
119                     nn.ConvTranspose2d(in_channels = 64,
                        ↪  out_channels = 32, kernel_size =
                        ↪  (4,4), stride = 2, padding = 1),
120                     nn.ConvTranspose2d(in_channels = 32,
                        ↪  out_channels = 16, kernel_size =
                        ↪  (4,4), stride = 2, padding = 1),
121                     nn.ConvTranspose2d(in_channels = 16,
                        ↪  out_channels = 8, kernel_size =
                        ↪  (4,4), stride = 2, padding = 1),
122                     nn.Conv2d(in_channels = 8,
                        ↪  out_channels = 2, kernel_size =
                        ↪  (1,1), stride = 1, padding = 0),
                        ↪  nn.Sigmoid())
```

```
123
124 # Selecting model
125 model = segnetbn
126
127 # Resetting model when we switch architectures
128 reset_model(model)
129
130 # Moving model to GPU
131 model.to(device)
132
133 # Check dimensions through network
134 # Credits:
    ↪ https://d2l.ai/chapter_convolutional-neural-networks/lenet.html
135 check_dims = False
136 if check_dims == True:
137     X = torch.rand(size = (num_batch, 3, 128, 128)).to(device)
138     for layer in model:
139         X = layer(X)
140         print(layer.__class__.__name__, 'output shape:\t',
            ↪ X.shape)
141
142 # Loss function(s)
143 bce = F.binary_cross_entropy
144 bce_logits = F.binary_cross_entropy_with_logits
145
146 it = 0
147 num_epochs = 300
148 l_rate = 0.001 # 0.001 works good
149
150 sgd_opt = optim.SGD(model.parameters(), lr = l_rate,
    ↪ weight_decay = 0, momentum = 0.0)
151
152 rmsp_opt = optim.RMSprop(model.parameters(), lr = l_rate,
    ↪ alpha = 0.99, eps = 1e-08, weight_decay = 0, momentum =
    ↪ 0.1, centered = False)
153
```

```
adam_opt = optim.Adam(model.parameters(), lr = l_rate, betas =
↪  (0.9, 0.999), eps = 1e-08, weight_decay = 0.0, amsgrad =
↪  False)

adad_opt = optim.Adadelta(model.parameters(), lr = 1.0, rho =
↪  0.9, eps = 1e-06, weight_decay = 0)

adag_opt = optim.Adagrad(model.parameters(), lr = 0.01,
↪  lr_decay = 0, weight_decay = 0, initial_accumulator_value
↪  = 0, eps = 1e-10)

adamax_opt = optim.Adamax(model.parameters(), lr = 0.002,
↪  betas = (0.9, 0.999), eps = 1e-08, weight_decay = 0)

asgd_opt = optim.ASGD(model.parameters(), lr = 0.01, lambd =
↪  0.0001, alpha = 0.75, t0 = 1000000.0, weight_decay = 0)

#sched = torch.optim.lr_scheduler.ExponentialLR(optr, gamma,
↪  last_epoch = -1, verbose = False)

optr = adam_opt
dscore = []
bcetrain = []
bcetest = []

for epoch in range(num_epochs):
    model.train()
    for xbatch, ybatch in training_loader:
        optr.zero_grad()

        prediction = model(xbatch)
        pred = prediction.mean(1) # float32, ybatch float64
        #pred = prediction.amax(1)

        bce_l = bce_logits(pred, ybatch)
        bcetrain.append(bce_l.item())

        bce_l.backward()
```

```
        optr.step()
        #sched.step()

        model.eval()
        with torch.no_grad():
            testpreds = model(xtest)
            testpred = testpreds.mean(1)
            #estpred = testpreds.amax(1)

            bce_test = bce_logits(testpred, ytest)
            bcetest.append(bce_test.item()) # ytest is
            ↪  float64, testpred is float32

            dice_score = dsc(testpred.double(), ytest)
            dscore.append(dice_score.item())

            if epoch == (num_epochs - 1):
                te_pred = testpred
                test_batch = ytest

                tra_pred = pred
                train_batch = ybatch

        it += 1
    print("Epoch %s/%s" % (epoch + 1, num_epochs))

plt.figure(1)
train, = plt.plot(bcetrain, 'r')
test, = plt.plot(bcetest, 'b')
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Training and testing loss")
plt.legend([train, test], ['Train loss', 'Test loss'])
plt.annotate("Final train loss: %s" % (bcetrain[-1]) ,xycoords
↪  = 'figure fraction', xy = (0.25,0.7))
plt.annotate("Final test loss: %s" % (bcetest[-1]), xycoords =
↪  'figure fraction', xy = (0.25,0.75))
print("Final training loss: %s." % bcetrain[-1])
```

```
print("Final testing loss: %s." % bcetest[-1])
plt.savefig("bce_loss_plots", dpi = 500)

num = 0
plt.figure(2)
plt.imshow(tra_pred[num, :, :].detach().cpu(), cmap = 'jet')
plt.title('Training prediction')
plt.colorbar()
plt.savefig("train_pred", dpi = 500)
plt.figure(3)
plt.imshow(train_batch[num, :, :].detach().cpu(), cmap =
↪  'jet')
plt.title('Training mask')
plt.colorbar()
plt.savefig("train_mask", dpi = 500)
plt.figure(4)
plt.plot(dscore, 'b')
plt.xlabel("Iteration")
plt.ylabel("DSC")
plt.savefig("dice_plot", dpi = 500)
print("Final dice coefficient: %s." % dscore[-1])

ind = 10
plt.figure(5)
plt.imshow(te_pred[ind, :, :].detach().cpu(), cmap = 'jet')
plt.title('Testing prediction')
plt.colorbar()
plt.savefig("test_pred_1", dpi = 500)
plt.figure(6)
plt.imshow(test_batch[ind, :, :].detach().cpu(), cmap = 'jet')
plt.title('Testing mask')
plt.colorbar()
plt.savefig("test_mask_1", dpi = 500)

ind = 20
plt.figure(7)
plt.imshow(te_pred[ind, :, :].detach().cpu(), cmap = 'jet')
plt.title('Testing prediction')
```

```
plt.colorbar()
plt.savefig("test_pred_2", dpi = 500)
plt.figure(8)
plt.imshow(test_batch[ind, :, :].detach().cpu(), cmap = 'jet')
plt.title('Testing mask')
plt.colorbar()
plt.savefig("test_mask_2", dpi = 500)
```