Deep Learning for Image Analysis
DL4IA – Report for Assignment III

Student Viktor Öhrner

May 10, 2021

1 Introduction

This exercise is divided into two part. The first one deals with creating a convoluted network for classifying the number of images and the second part deals with segmentation of cancer cells. In the first part the MNIST data set is used for all parts. The data set consists of 60000 training images, 10000 test images. The size of the images is $28 * 28$ grey scale pixels and there are ten different classes.

Exercise 1.1 *Implement a fully connected network* In this exercise PyTorch is used to implement a fully connected network. The network is made up of $28 * 28 = 784$ input nodes, two hidden layers of size 80 and 40 nodes and ends with a softmax function. The cost function used was cross entropy. The net was trained using mini-batches of size 1000, 1000 iterations, and a learning rate of 0.01.

Exercise 1.2 *Implement a convolutional network* In this part a convoluted network is implement to classify the images. It consists of 12 layers in total with three of them being convoluted layers each followed by a ReLu activation function. The net also have two max pooling layers of size 2*2 and ends with a dense net followed by a softmax layer. The loss function used is cross entropy.

Exercise 1.3 *Swapping ReLu and max pooling* In this part of the exercise all sequences of a ReLu layers followed by a max pooling layer is swapped so that the max pooling layer is first followed by the ReLu layer.

Exercise 1.4 *Adam as optimizer* Exchange the optimizer from stochastic gradient descent to Adam which is an optimizer that degrades the learning rate.

Exercise 1.5 *Variations* There was also other variations done to the net, starting from the net generated in part 1.4. The following changes was tried:

- Batch normalization before the third convolutional layer.
- Add another convoluted layer of size $64 \times 3x3x32$ followed by a max pooling of size $2x2$ and padding 1 into a ReLu.
- Changed the activation functions from ReLu to tanh.
- Unmake the changes from part 1.3.

Exercise 2.1 *Segmentation of the Warwick biomedical data set* Re-purpose a classification CNN to a network that can perform segmentation. In this section a data set called WARWICK containing microscopy images of colon cancer is used. The images in the set have a size of $128x128$ and two channels while the label images have the same size but one channel with two different classes '0' or '1'.
The net is re-purposed by adding transposed convolutional layers to the end of a classification network. The network that is modified is the net used in exercise 1.3. To start with the dense layer at the end is removed then two up-scaling layers are added so that the original image size is restored, each up-scaling

layer is followed by a ReLu layer. The result is two channels of size $128x128$, the reason that there are two channels is because there are two different classes that the segmented image can have.
The learning-rate is 0.01, batch size is 5, 5000 iterations where performed resulting in $5000 * 5/85 \approx 294$ epochs. The loss function used is cross entropy.

Exercise 2.2 *Variations* The net from the previous exercise was modified. To determine what modification was best the training data was split into validation and training sets. The size of the validation set is 20% and the new test set is 80%. The validation set was used to validate the performance of different hyper-parameters of the net, then the test set is used to estimate the performance of the net. The modifications are to the size and shape of the net, a L2 regularization coefficient is added, and the test data is augmented. The performance of these modifications was tested separately and then all put together. The different size changes was to increase/decrease either the number of layer, numbers or feature maps. The equation for L2 regularization can be seen in equation (2), the set of values tested for λ was [0.01, 0.005, 0.001]. The augmentations to the data consists of adding a rotating version of random images to the training set after it had been split into training and validation. For each image there was a 25% that is was rotated 90°, 180°, or 270° clock wise.

2 Mathematical exercises

The loss function used to train the net is the cross entropy function. For a data point L_i the loss is calculated as:

$$L_i = ln(\sum_{l=1}^{M} e^{z_{il}}) - \sum_{m=1}^{M} \tilde{y}_{im} z_{im} \tag{1}$$

Where M is the number of classes, z_{il} is the prediction for the class l, and $\tilde{y}_{im}$ is the one-hot encoding of the true label y_i.

Exercise 2.2 *Variations* In this exercise a L2 regularization coefficient is added to the loss function. The new loss function is defined as:

$$L_i = ln(\sum_{l=1}^{M} e^{z_{il}}) - \sum_{m=1}^{M} \tilde{y}_{im} z_{im} + \lambda \sum_{j=0}^{M} W_j^2 \tag{2}$$

3 Code exercises

Exercise 1.1 *Implement a fully connected network* The network was implemented as a class called 'Net' with an initialize and forward function that can bee seen bellow.

- The *initialize* function is used to define the size of the linear part of the net.
- The *forward* function takes a data set input x and makes a prediction. In this function the network is defined by implementing the non-linear function and softmax layer.

```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 80)
        self.fc2 = nn.Linear(80, 40)
        self.fc3 = nn.Linear(40, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Exercise 1.2 *Implement a convolutional network* The network consists of 12 layers see the code of the code bellow for the architecture. The first layer defined on line 4 is a convolutional layer with one and 8 feature maps as output, the second and third layer is defined on line 11 starting with a ReLu function followed by max pooling of size $2x2$ with padding 2. This pattern of convoluted layer followed by a ReLu into a max pooling is repeated in line 5 and 12. The a last convoluted layer into a ReLu before a fully connected layer the is defined on line 7 and used on line 15 and then the result is run through a softmax function. This results in the first convoluted layer having $8 * 3 * 3 = 72$ weights, second layer having $16 * 8 * 3 * 3 = 1152$ weights, the last convoluted layer having $32 * 16 * 3 * 3 = 4608$ weights, and the last fully connected layer have $1568 * 10 = 15680$ weights. All those weights adds up to a total of 21512. Compare this to the fully connected layer made in the previous exercise that had a total of 66320 weights. During training a batch size of 80 was used, 2000 iterations resulting in 16 epochs, and a learning rate of 0.01 was used. The optimization function used was stochastic gradient descent with a momentum of 0.9.

```
1  class Net(nn.Module):
2      def __init__(self):
3          super(Net, self).__init__()
4          self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=(1, 1), padding_mode='reflect')
5          self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=(1, 1), padding_mode='reflect')
6          self.conv3 = nn.Conv2d(16, 32, kernel_size=3, padding=(1, 1), padding_mode='reflect')
7          self.fc = nn.Linear(1568, 10)
8  
9      def forward(self, x):
10         in_size = x.size(0)
11         x = F.max_pool2d(F.relu(self.conv1(x)), 2, stride=(2, 2))
12         x = F.max_pool2d(F.relu(self.conv2(x)), 2, stride=(2, 2))
13         x = F.relu(self.conv3(x))
14         x = x.view(in_size, -1)
15         x = self.fc(x)
16         return x
```

Exercise 1.3 *Swapping ReLu and max pooling* In this exercise some layers are switched around. The two first instances of a ReLu followed by a max pooling have been swapped so that it is the ReLu that follows the max pooling. The code from the previous exercise was used but changes in the `forward` function was made, see code bellow. On line 3 and 4 the order of `F.relu` and `F.max_pool` was changed.

```
1      def forward(self, x):
2          in_size = x.size(0)
3          x = F.relu(F.max_pool2d(self.conv1(x), 2, stride=(2, 2)))
4          x = F.relu(F.max_pool2d(self.conv2(x), 2, stride=(2, 2)))
5          x = F.relu(self.conv3(x))
6          x = x.view(in_size, -1)
7          x = self.fc(x)
8          return x
```

Exercise 1.4 *Adam as optimizer* For this exercise the same code was used as the previous but the optimizer was changed from SGD to Adam. See the code bellow on line 1 for how Adam was defined. Where net is the net and `lr` is the learning rate set to 0.01

```python
optimizer = optim.Adam(net.parameters(), lr)
```

Exercise 1.5 *Variations to the net* For the first variation a batch normalization is implemented see the code bellow. On line 8 the batch normalization is defined as having 16 channels, and on line 15 the batch normalization is performed.

```python
 1  class Net(nn.Module):
 2      #Added batch normalization
 3      def __init__(self):
 4          super(Net, self).__init__()
 5          self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=(1, 1), padding_mode='reflect')
 6          self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=(1, 1), padding_mode='reflect')
 7          self.conv3 = nn.Conv2d(16, 32, kernel_size=3, padding=(1, 1), padding_mode='reflect')
 8          self.bn1 = nn.BatchNorm2d(16)
 9          self.fc = nn.Linear(1568, 10)
10
11      def forward(self, x):
12          in_size = x.size(0)
13          x = F.relu(F.max_pool2d(self.conv1(x), 2, stride=(2, 2)))
14          x = F.relu(F.max_pool2d(self.conv2(x), 2, stride=(2, 2)))
15          x = self.bn1(x)
16          x = F.relu(self.conv3(x))
17          x = x.view(in_size, -1)
18          x = self.fc(x)
19          return x
```

For the second alternative another layer is added, creating 64 features maps of size $3x3$ before the final dense layer. This means that another max pooling layer was implemented.

```python
class Net(nn.Module):
    # Extra convolution layer
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=(1, 1), padding_mode='reflect')
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=(1, 1), padding_mode='reflect')
        self.conv3 = nn.Conv2d(16, 32, kernel_size=3, padding=(1, 1), padding_mode='reflect')
        self.conv4 = nn.Conv2d(32, 64, kernel_size=3, padding=(1, 1), padding_mode='reflect')
        self.fc = nn.Linear(576, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(F.max_pool2d(self.conv1(x), 2, stride=(2, 2)))
        x = F.relu(F.max_pool2d(self.conv2(x), 2, stride=(2, 2)))
        x = F.relu(F.max_pool2d(self.conv3(x), 2, stride=(2, 2)))
        x = F.relu(self.conv4(x))
        x = x.view(in_size, -1)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)
```

For the third version the ReLu have been changed to tanh activation functions. In the instance of the first layer the code looks like:

```
x = F.hardtanh(F.max_pool2d(self.conv1(x), 2, stride=(2, 2)))
```

For the final version have the max pooling and ReLu layers switch, see exercise 1.2 to see how that would look like.

Exercise 2.1 *Segmentation* The net is defined as can be seen bellow. The two first convolutional layers are each followed by a max pooling and a ReLu layer. The there is a third layer followed by a ReLu layer. After that the up-scaling starts. The first transposed convolutional layer is defined on line 7 and it takes 32 channels and outputs 16 channels. When the net is trained for the Warwick data set where the image sizes are 128x128 the feature maps supplied to this first transposed convolutional layer will have the size 32x32, because the layer have a kernel of size 4, a stride of 2, and a padding of 1 the resulting feature maps will have size 64. The transpose convolutional layer is followed by a ReLu layer seen on line 15. This is then repeated again so that 8 channels of size 128x128 are generated. From these feature maps a 1x1 convolution is applied, defined on line 9 and used on line 17 this will output two channels of the right size.

```
1  class Net(nn.Module):
2      def __init__(self):
3          super(Net, self).__init__()
4          self.conv1 = nn.Conv2d(2, 8, kernel_size=3, padding=(1, 1), padding_mode='reflect')
5          self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=(1, 1), padding_mode='reflect')
6          self.conv3 = nn.Conv2d(16, 32, kernel_size=3, padding=(1, 1), padding_mode='reflect')
7          self.tconv1 = nn.ConvTranspose2d(32, 16, kernel_size=4, stride=2, padding=1)
8          self.tconv2 = nn.ConvTranspose2d(16, 8, kernel_size=4, stride=2, padding=1)
9          self.convFinal = nn.Conv2d(8, 2, kernel_size=1, stride=1)
10
11     def forward(self, x):
12         x = F.relu(F.max_pool2d(self.conv1(x), 2, stride=(2, 2)))
13         x = F.relu(F.max_pool2d(self.conv2(x), 2, stride=(2, 2)))
14         x = F.relu(self.conv3(x))
15         x = F.relu(self.tconv1(x))
16         x = F.relu(self.tconv2(x))
17         x = F.relu(self.convFinal(x))
18         return x
```

The training and evaluation of the network is done in a for-loop. On line 2 a tensor of random indices are generated so each batch have a random set of images. On line 4 and 5 the batches for the iteration is constructed. Then before the training is done the gradient is zeroed out on line 6. A forward pass is performed on the input batch and the result is saved on line 7. The output of the net is of size [batch_size, 2, 128, 128]. Then the loss is calculated on the next line, here `Y_batch` is of size [batch_size, 128, 128]. On line 9 the gradient is calculated and on the next line the net is updated. On line 13 the evaluation using the test data starts by making a prediction, the output from the net is of size [65, 2, 128, 128].

```
1  for epoch in range(EPOCHS):
2      random_index = torch.randperm(X_train.size(0))
3      for iteration in range(ITERATION_PER_EPOCH):
4          X_batch = X_train[random_index[iteration*batch_size:(iteration+1)*batch_size],:,:,:]
5          Y_batch = Y_train[random_index[iteration*batch_size:(iteration+1)*batch_size],:,:]
6          net.zero_grad()
7          output = net(X_batch.float())
8          loss = criterion(output, Y_batch)
9          loss.backward()
```

```
        optimizer.step()
        loss_train[epoch*ITERATION_PER_EPOCH+iteration] = loss.item()
        output = output.argmax(axis=1)
        output_test = net(X_test.float())
        loss_test_it = criterion(output_test.squeeze(), Y_test)
        loss_test[epoch*ITERATION_PER_EPOCH+iteration] = loss_test_it.item()
        output_test = output_test.argmax(axis=1)
        dice_train[epoch*ITERATION_PER_EPOCH+iteration] = calc_dice_coef(output, Y_batch)
        dice_test[epoch*ITERATION_PER_EPOCH+iteration] = calc_dice_coef(output_test, Y_test)
```

Exercise 2.2 *Variations* The L2 regularization was implemented by using the `AdamW` function. For the instance of having λ set to 0.001 the code looks like.

```
    optimizer = optim.AdamW(net.parameters(), lr, weight_decay=0.001)
```

4 Results

Exercise 1.1 *Implement a fully connected network* The accuracy and loss of the training and test data over the iterations can be seen in Figure 1 and Figure 2. The final accuracy of the test set was 0.97 and the time for the training was 1 min and 9 seconds. Compare this to the previous assignment where the same net was made using numpy which took 6 min and 50 seconds to train and a accuracy of 0.902. The model appears to be a bit over-fitted and the optimal net seems to appear around iteration 800. This can be seen in the loss figure where the training loss decreases around iteration 800 but then increases again.

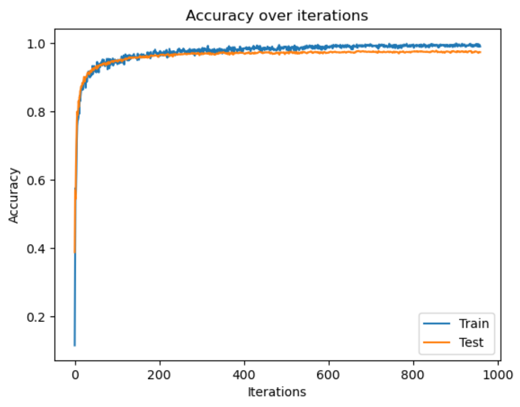

Figure 1: Accuracy over iterations

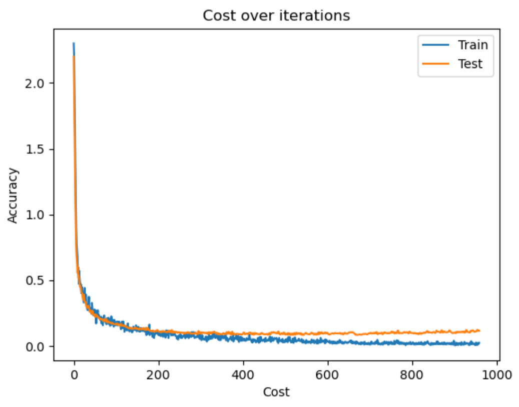

Figure 2: Cost over iterations

Exercise 1.2 *Implement a convolutional network* The trained model was able to reach an accuracy of 0.982 on the test data. The accuracy on train and test data over the training iterations can be seen in Figure 3 and cost in Figure 4. The model took 25 min to train. Judging from the graphs the model does not seem over-fitted since the test accuracy does not decrease and the loss does not start to increase. However I don't think it is valid to say the model is under fitted since the reason for the train and test accuracy is similar because it is so close to 100%.

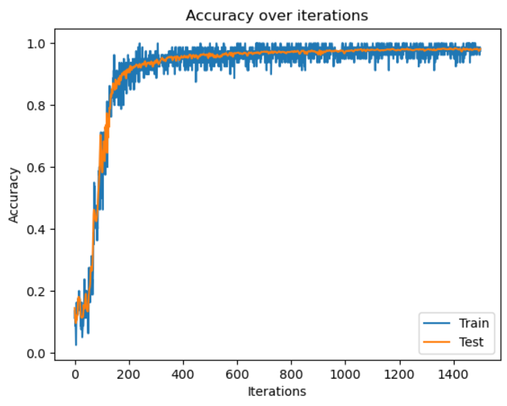

Figure 3: Accuracy over iterations

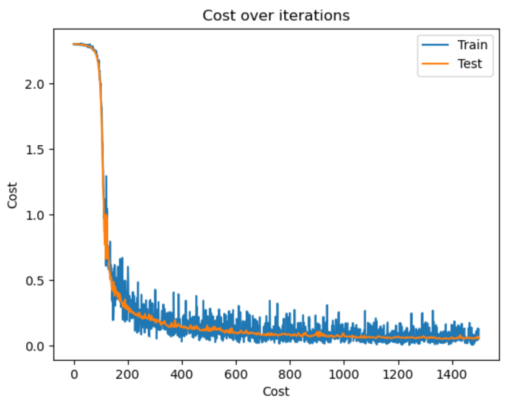

Figure 4: Cost over iterations

Exercise 1.3 *Swapping ReLu and max pooling* The ReLu and max pooling where swapped such that the max pooling was followed by a ReLu function. The accuracy on the testing data was 0.979, this is not a huge difference to the previous exercise that had a accuracy of 0.982. In Figure 5 the accuracy over iteration can be seen and in Figure 6 the cost can be seen. Compared to the previous exercise it seems like the cost for this model goes down faster. The model took 20 min, this is 5 min or 20% faster than having the ReLu before the max pool layer. From this there seems to be a difference in how fast the model is trained but not a major drop of in the performance. The reason that the performance is similar is because the order that these two layers are applied does not change the outcome. Regarding the fitting of the model it does not over-fit the data and similar to the previous exercise it would be hard to claim that the model is under-fitting since the model's accuracy is so close to 100%.

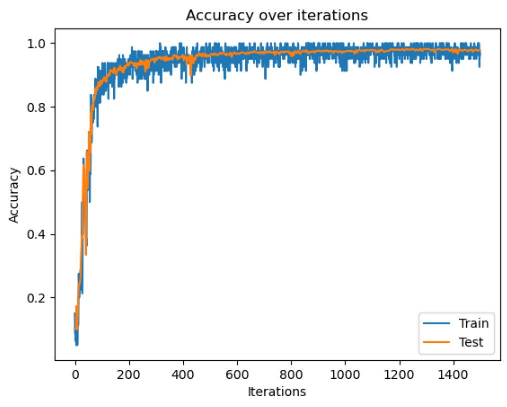

Figure 5: Accuracy over iterations

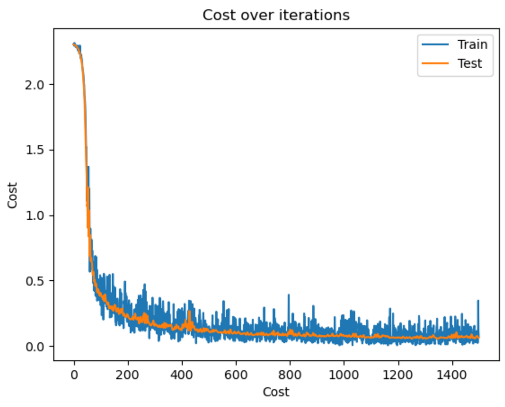

Figure 6: Cost over iterations

Exercise 1.4 *Adam as optimizer* The model took 22 min to train which is 2 min slower than when SGD was used as optimizer. An accuracy of 0.982 was achieved and a final cost of 0.0509. The accuracy and cost of the test and training data can be seen in Figure 7 respectively Figure 8. The noticeable change is that the training time increased by 2 min, so Adam seems to take require more time to train the network. However the accuracy is bumped back up to 0.982, so it seems that there is a payoff to the increased training time with an increase in the accuracy.

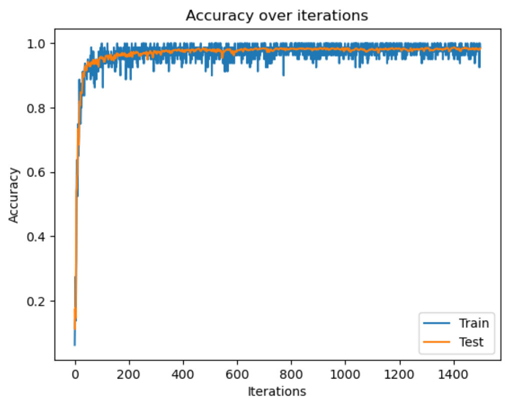

Figure 7: Accuracy over iterations

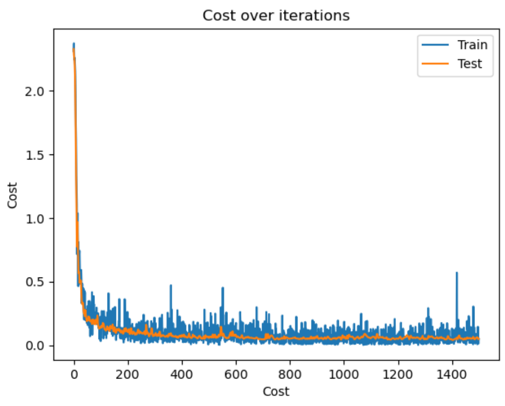

Figure 8: Cost over iterations

Exercise 1.5 *Changes to the network* The accuracy on the test data for the different modifications where as following:

- Batch normalization: 0.978
- Extra convoluted layer: 0.964
- tanh activation: 0.971
- Swapped ReLu and max pooling: 0.984

The best performance was from when the ReLu and max pooling swapped positions so that the ReLu is applied before the max pooling. While the accuracy was improved by 0.002 percentage points compared to exercise 1.4 the time it took to train the model was 47 min, more than double the time in exercise 1.4. The accuracy and cost over each iteration can be seen in Figure 9 and Figure 10. The model does not appear to be over-fitted, but stopping 500 iterations earlier would not worsen the performance.

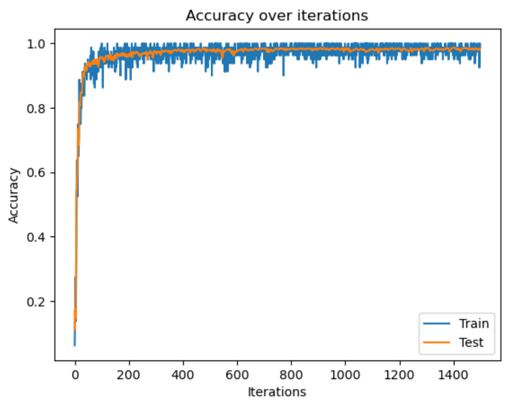

Figure 9: Accuracy over iterations

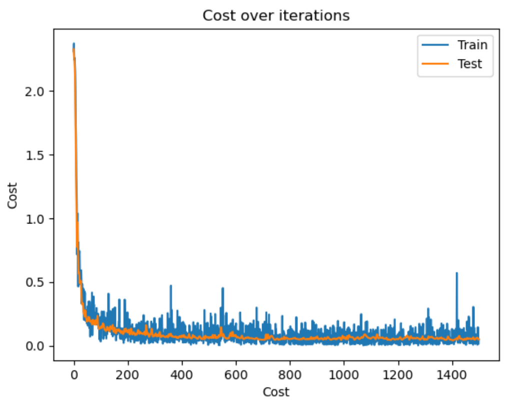

Figure 10: Cost over iterations

The following four images where wrongly classified, see Figure 11.

The confusion matrix can be seen bellow. The top row represents the true label and the left column represents the guess made by the net. From this it is observable that the number seven is the most common number to be a miss classification.

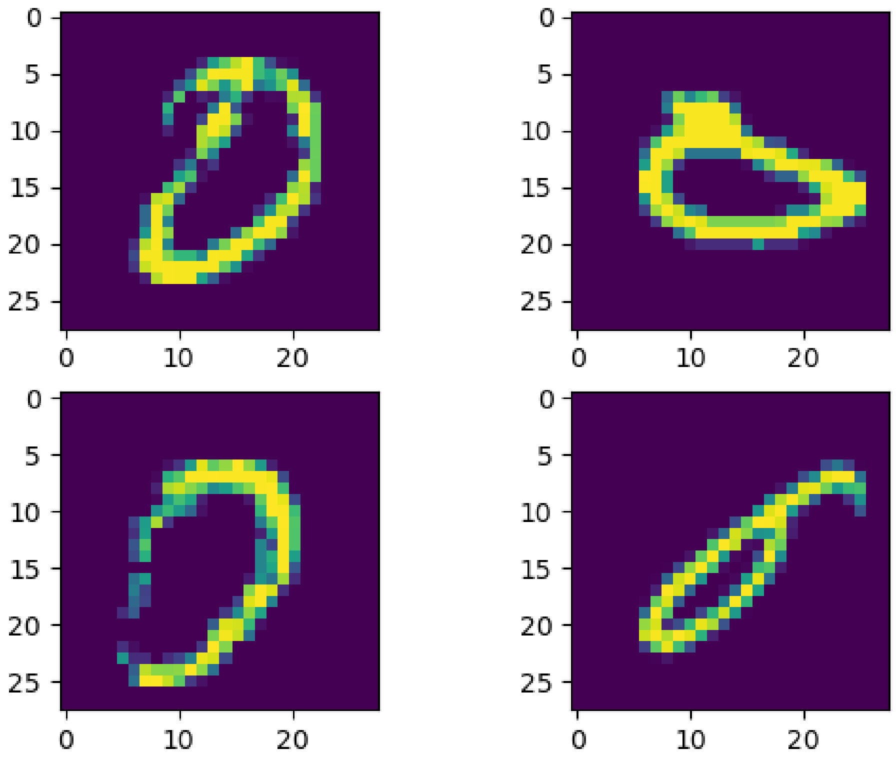

Figure 11: Top left) guess was 2 correct label is 0. Top right) guess was 6 correct label is 0. Bottom left) guess was 9 correct label is 0. Bottom right) guess as 8 correct label is 0.

.	0L	1L	2L	3L	4L	5L	6L	7L	8L	9L
0G	974	0	2	0	0	0	2	0	1	1
1G	0	1134	0	0	0	0	1	0	0	0
2G	1	4	1024	0	0	0	1	0	2	0
3G	0	1	9	987	0	4	0	1	8	0
4G	0	0	0	0	967	0	2	0	5	8
5G	3	0	0	11	0	864	2	0	9	3
6G	4	3	0	0	2	4	939	0	6	0
7G	0	11	17	2	1	0	0	983	3	11
8G	4	0	2	0	0	0	1	0	965	2
9G	3	2	2	0	6	1	0	3	14	987

Exercise 2.1 *Segmentation of Warwick biomedical data set* The dice coef. and loss can be seen is Figure 12 and Figure 13. The final Dice coef. was 0.789. A poorly made guess by the net can be seen in the top left corner of Figure 14, in the top right corner is the labeled image, and the input images can be seen in the bottom half of the figure. I believe that the reason for the net to perform bad on this image is because the texture of stained cancer and background is more similar compared to the other images. There is also a patch in the bottom right corner that is missing any sample and I believe this is messing with the model.

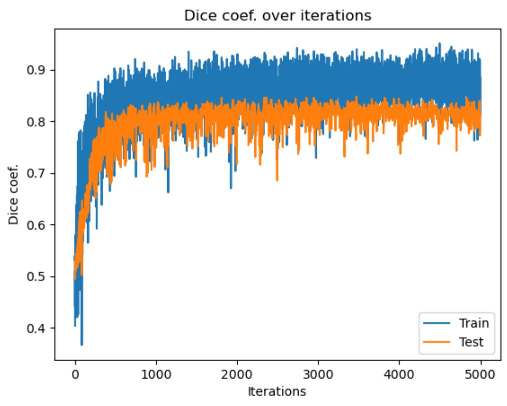

Figure 12: Dice coef. over iterations

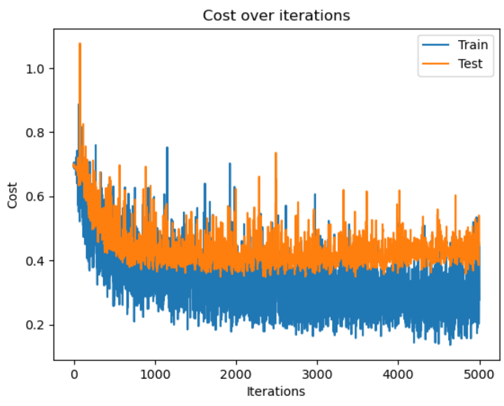

Figure 13: Cost over iterations

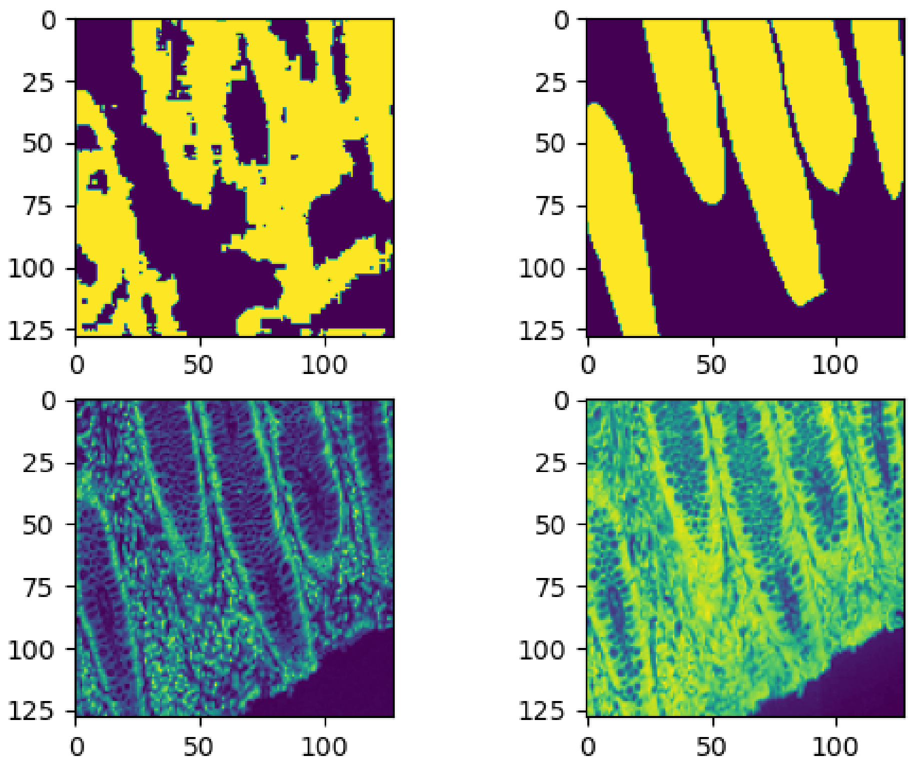

Figure 14: Top left: Guess made by the net. Top right: Labelling from data set.

Exercise 2.2 *Variations* While none of the variations implemented by themselves had a major effect, When they where implemented together a better dice coef. was achieved for both the test and validation set. The best performing net have the following structure:

1. Convolution - 10 times 3x3x2 convolution with stride 1 and padding 1 Max Pooling - 2x2 max pooling with stride 2
2. ReLu - Non-linear
3. Convolution - 20 times 3x3x10 convolution with stride 1 and padding 1 Max Pooling - 2x2 max pooling with stride 2

4. ReLu - Non-linear
5. Convolution - 40 times 3x3x20 convolution with stride 1 and padding 1
6. ReLu - Non-linear
7. Transpose Convolution - 20 times 4x4*40 with stride 2 and padding 1
8. ReLu - Non-linear
9. Transpose Convolution - 10 times 4x4*20 with stride 2 and padding 1
10. Convolution - 2 times 1x1x10 convolution with stride 1
11. Softmax

The resulting graphs of the dice coef. and cost can be seen in Figure. 15, the final dice coef. on the validation set was 0.851 and 0.878 for the training set, comparing this to the performance on the test set which gave a Dice coef. of 0.751. This big difference between the validation set and test set is chocking to me and it seems that the model is over fitted. I guess this is a result of having to many rotated images in the training set from the data augmentation.

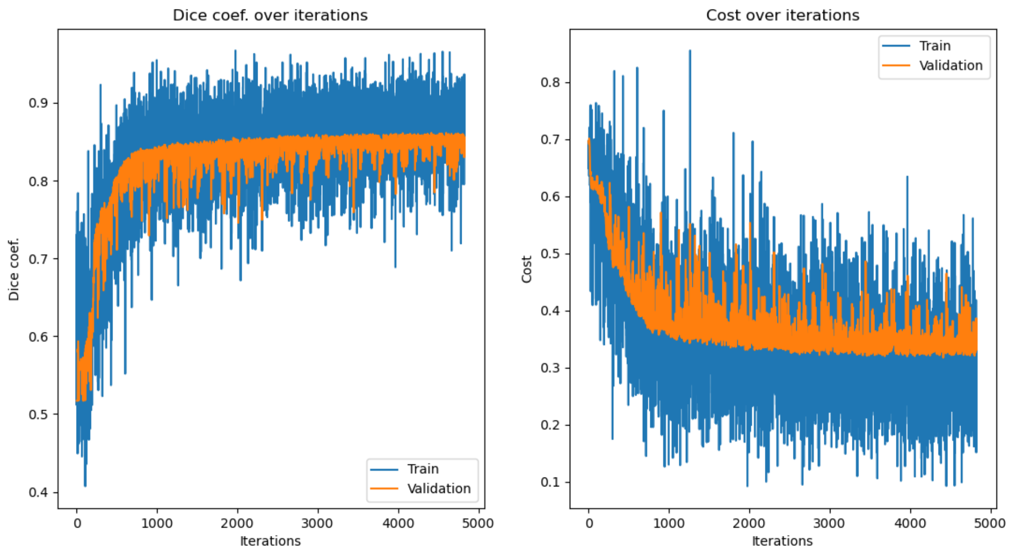

Figure 15: Left: Dice coef. Right: Cost.