Deep Learning for Image Analysis
Assignment 3
1MD120 61612 VT2024

Georgios Tsouderos

June 2024

Contents

1 MNIST Classification

1.1 Exercise 1.1

1.1.1 a

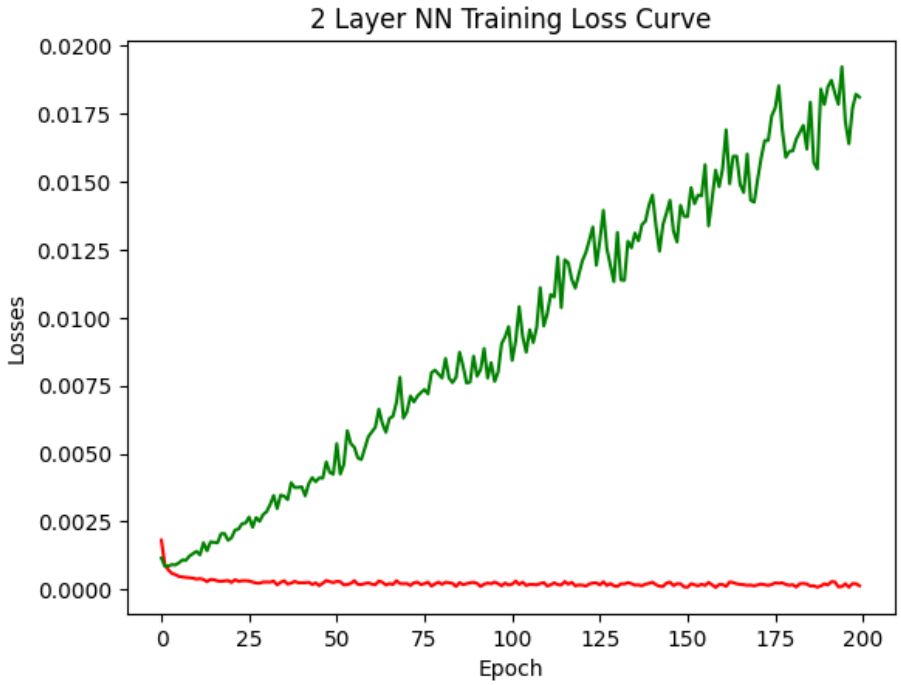
We observed that with the addition of Adam optimizer, the convergence was much faster. However, even though the architecture of the neural network and the hyperparameters were the same, we didn't reach the same performance. Oddly, the implementation on Assignment 2 performed better (for the same hyperparameters).

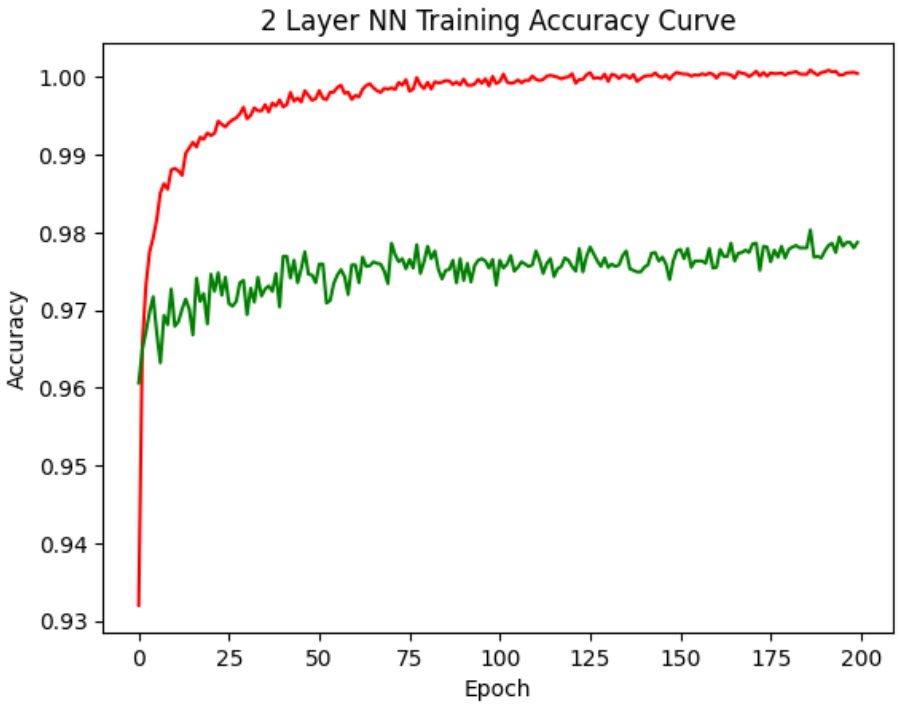
1.1.2 b

We trained both networks for 200 epochs, with a batch size of 128. The implementation in Assignment 2 was much faster, since the network would train in around 5 minutes. The Pytorch implementation took 27 minutes running on T4 GPU.

1.1.3 c

The red curve represents the train performance. The green curve is the test performance.





From the losses graph, we can see that there is definitely overfitting, since the loss for the training set is very low but on the other hand, the loss on the test set grows larger per epoch. The set of hyperparameters are not the correct ones.

We can see that the accuracy on the test set, even though it starts very high, doesn't improve much. In addition, the network seems to overfit since not only is there a great difference between training and test accuracy but also the training accuracy reaches 1.

1.2 Exercise 1.2

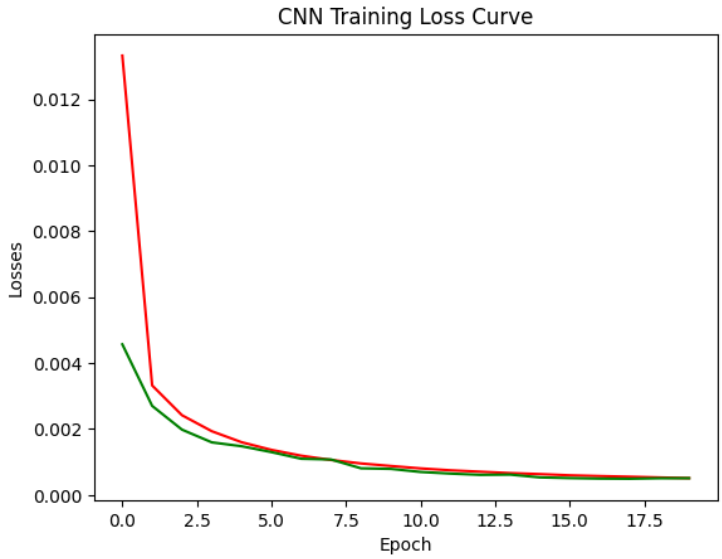
1.2.1 a

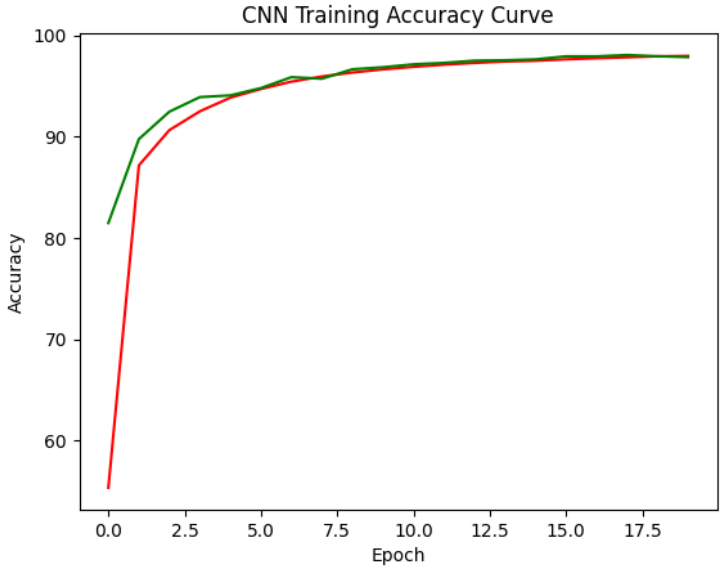
The only layers that have learnable parameters are the convolutional layers. The formula for calculating them is $(kernel\_size * input\_channels + bias) * output\_channels$. Also, every pooling layer, we reduce the size of the image by half. As a result, the size of the input image to the fully connected layer will be $7x7x32$. Thus, the number of learnable parameters for the whole network will be : $80 + 1168 + 4640 + 15690 = 21578$

For the previous implementation with the 2 layers, we will have $728*100+100$ for the first layer and then for the second layer $100 * 10 + 10$. In total, we have 79510 learnable parameters.

1.2.2 b

The red curve represents the train performance. The green curve is the test performance.

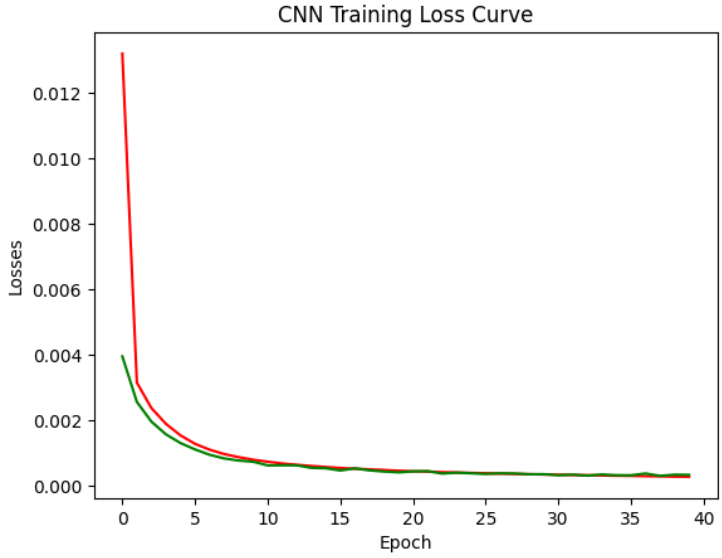


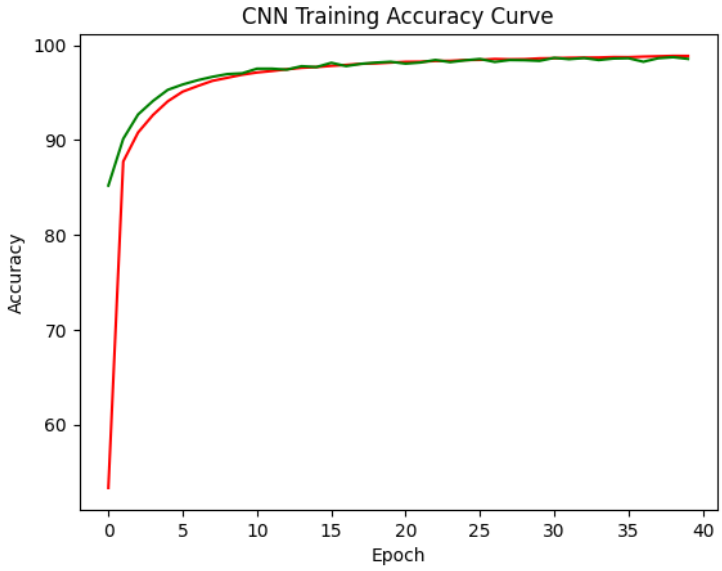


1.3 Exercise 1.3

1.3.1 a

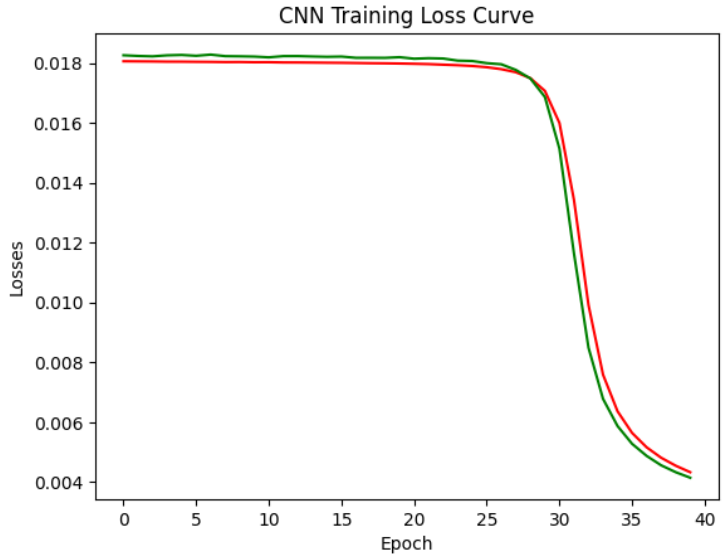
For 40 epochs, the previous implementation took 5 minutes to train, with a final test accuracy of 98.78%. After changing the order of operations, the training time was again 5 minutes, with a resulting accuracy of 98.57%.

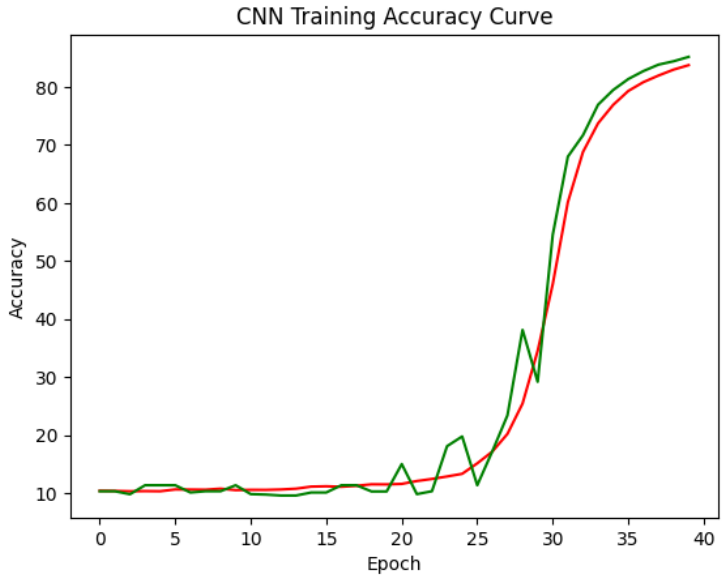




1.3.2 b-c

The training took again 5 minutes, with an accuracy score of 85.17%.



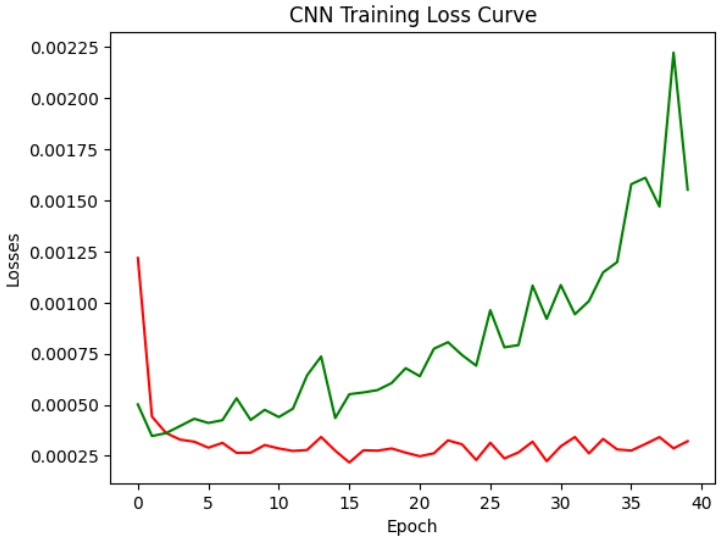


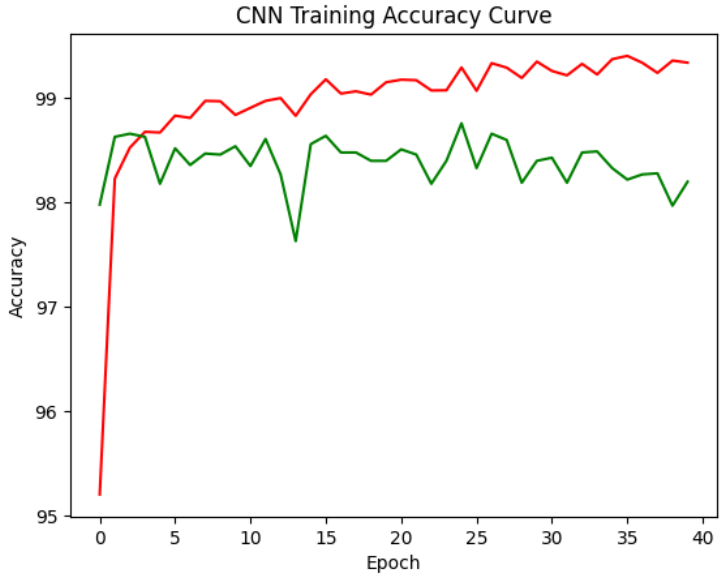
We observed that the convergence is much slower than the original versions. Accuracy started from 10%, slowing trending upwards. Probably a higher learning rate would improve convergence. The reason the above phenomenon is ob-

served has both to do with the order of operations and the switch from relu to sigmoid. Applying pooling first, reduces the size of the input and the resulting image consists primarily of positive numbers. As a result, relu has a weaker effect and the effect of non-linearity is less prominent. Switching from relu to sigmoid pushes the values in the range of 0-1. All together, we get a strong smoothing effect to the feature maps and smaller gradients. As a result, the learning process is slower but can be useful if the images are very noisy.

1.4 Exercise 1.4

Switching to an ADAM optimizer with epoch size of 40, the training takes again 5 minutes.



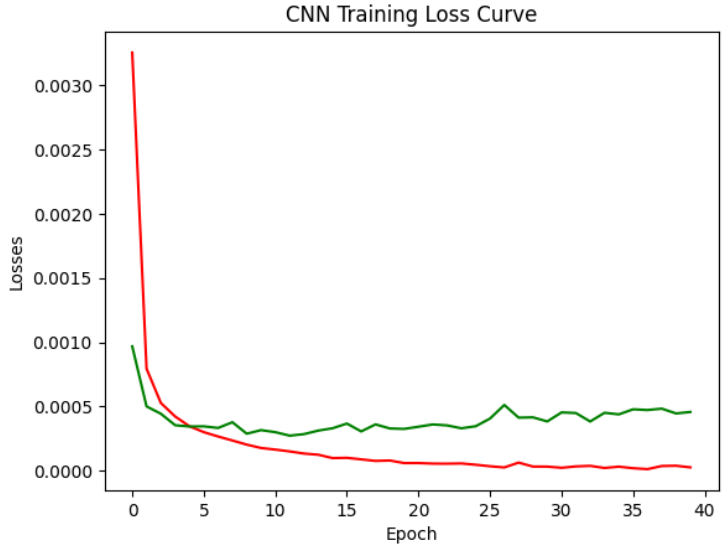


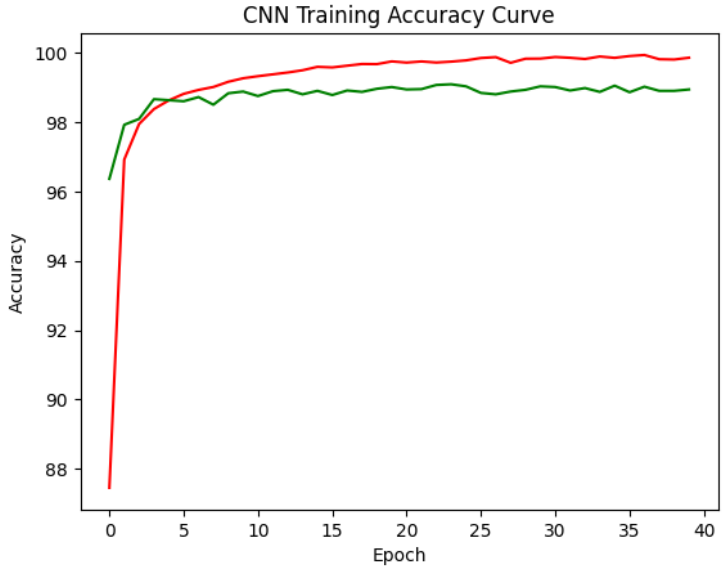
Oddly, the results don't seem to be better (98.2% accuracy) but overfitting is obvious. That gives us room to work with the hyperparameters and achieve better performance.

1.5 Exercise 1.5

1.5.1 Variation 1/3

For this variation, the learning rate was reduced to 0.001. This resulted in an accuracy score on the test set of 99%.

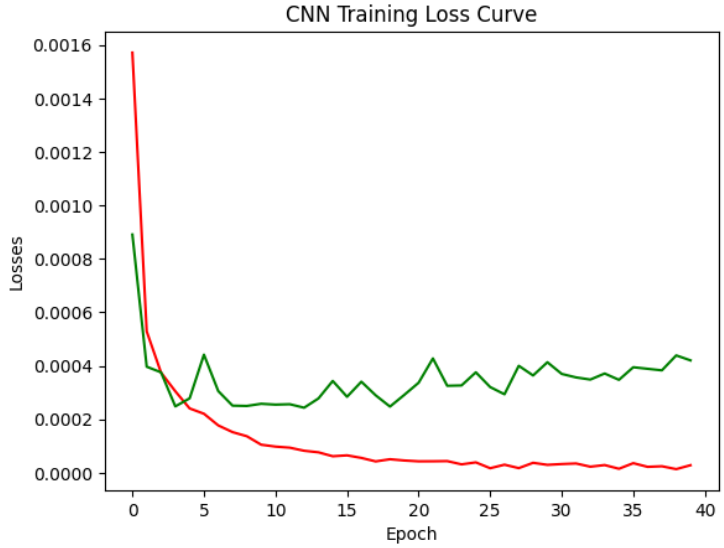


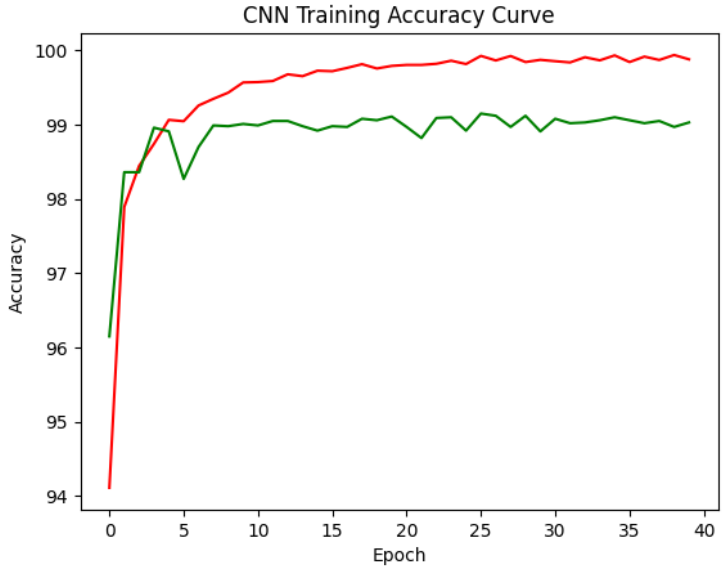


Already overfitting is reduced.

1.5.2 Variation 2/3

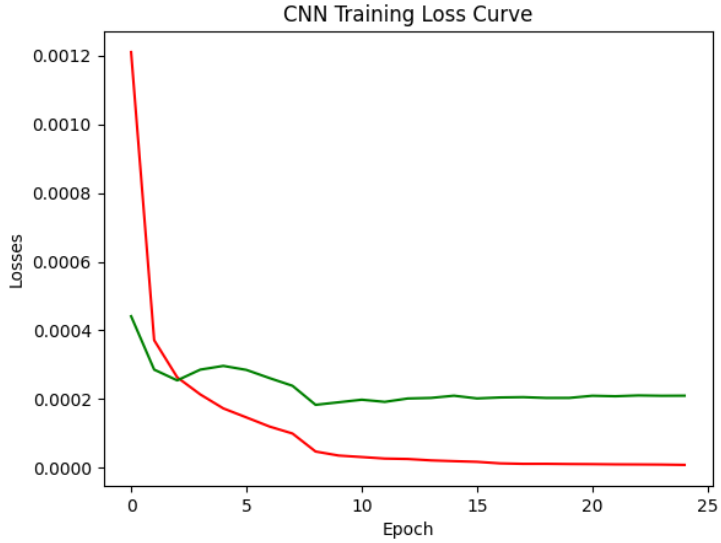
To further deal with the overfitting, we introduce batch normalization to the architecture of the model. The performance achieved was again 99/

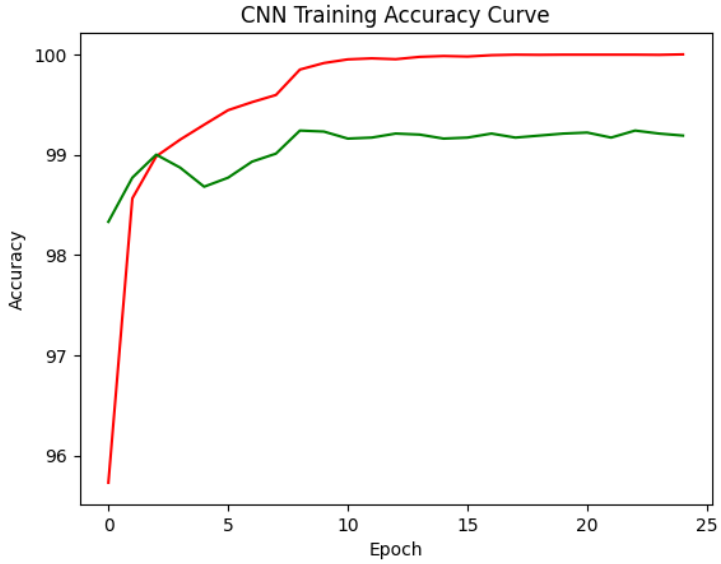




1.5.3 Variation 3/3

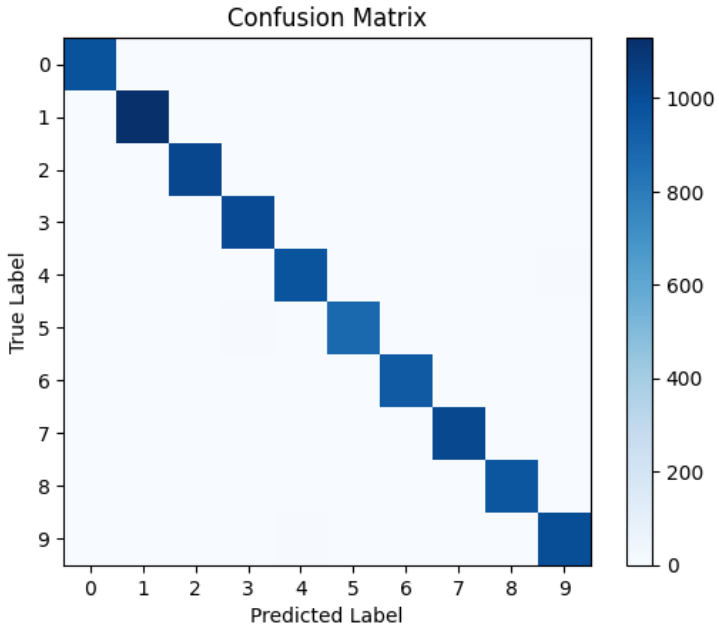
We introduce a learning rate scheduler to automatically reduce the learning rate according to the number of epochs. This increased the performance on the test set to 99.2%, which when it comes to such high accuracies values, is significant.



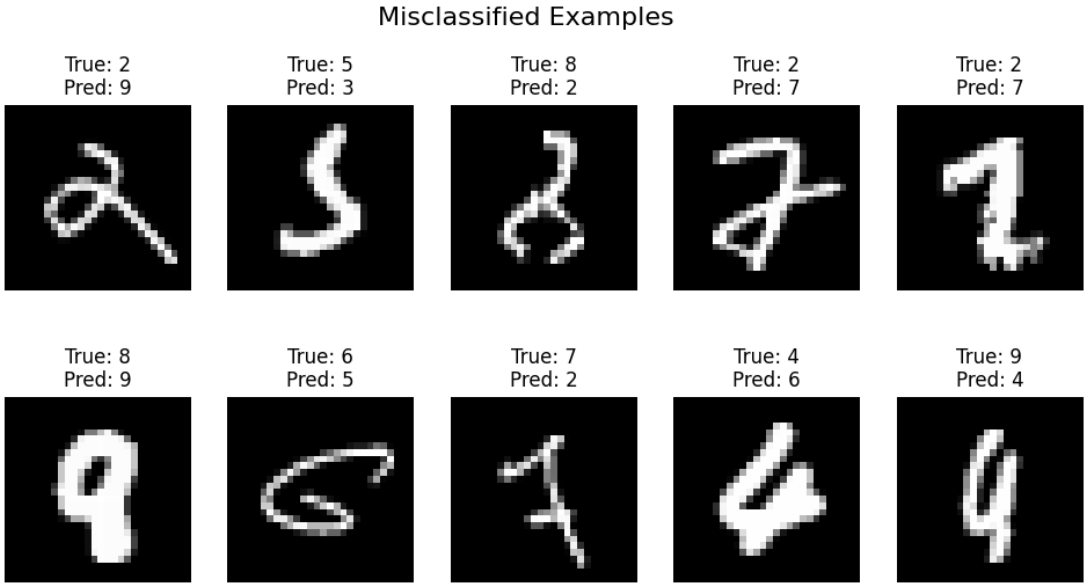


1.5.4 d

The best model was variant 3. The resulting confusion matrix is the following:


Confusion Matrix
True Label
Predicted Label
0
1
2
3
4
5
6
7
8
9
1000
800
600
400
200
0

1.5.5 e

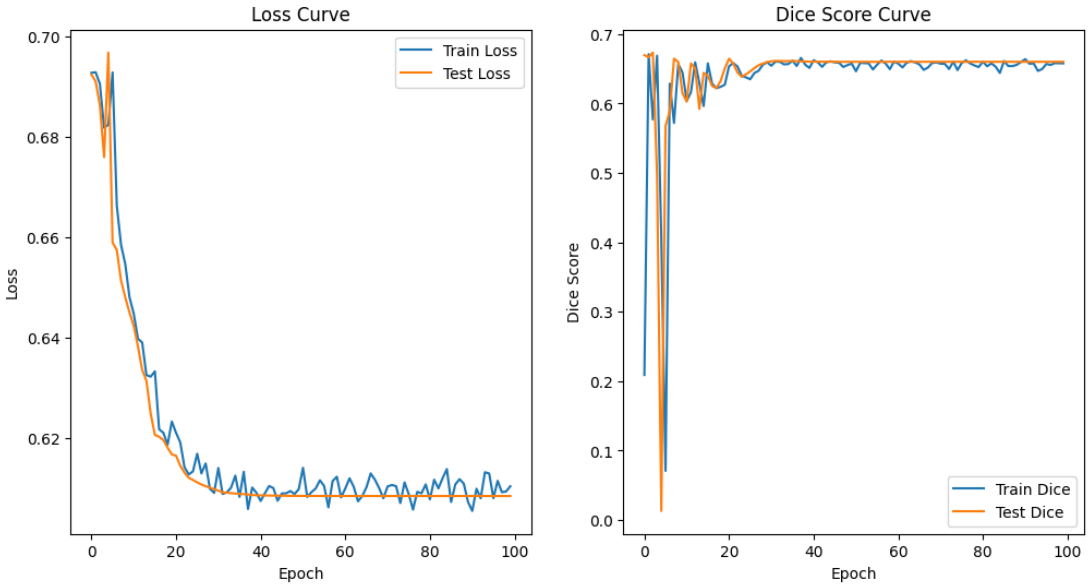

Misclassified Examples
True: 2
Pred: 9
True: 5
Pred: 3
True: 8
Pred: 2
True: 2
Pred: 7
True: 2
Pred: 7
True: 8
Pred: 9
True: 6
Pred: 5
True: 7
Pred: 2
True: 4
Pred: 6
True: 9
Pred: 4

2 Segmentation of the Warwick Biomedical Dataset

2.1 Exercise 2.1

2.1.1 c

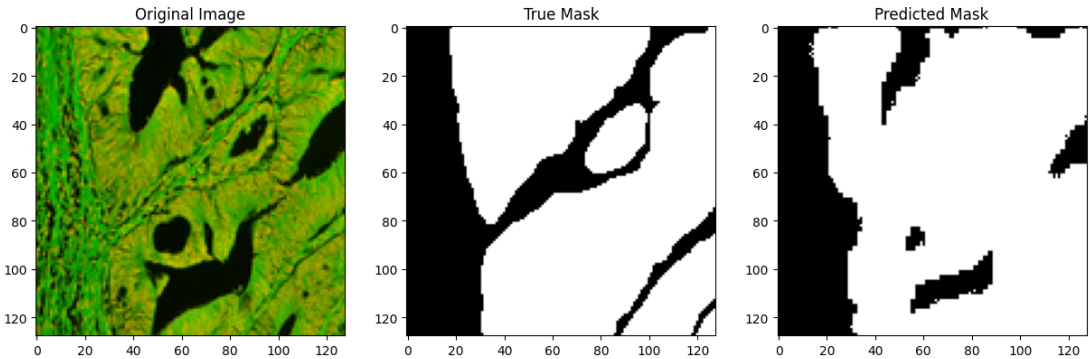
After training the model for 100 epochs with a learning rate of 0.01 and batch size 32, a dice score of 0.66 was achieved on the test set.



The model seems to be underfitting judging by how close the test and the train performance are. Also, the dice score seems to be quite low.

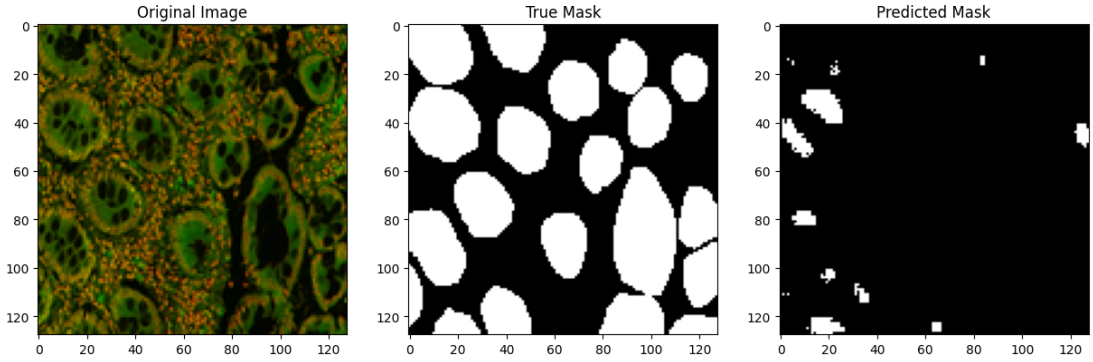
We can visually inspect what the model learned by showing an image, its true mask and the learned mask by the model.

In the following case, we have a dice score of 0.87:



We can see that the predicted mask has the general structure of the ground truth one, but missing some details. However, the area covered is what gives this high dice score.

On the other hand, we have an example with a dice score of 0.1:

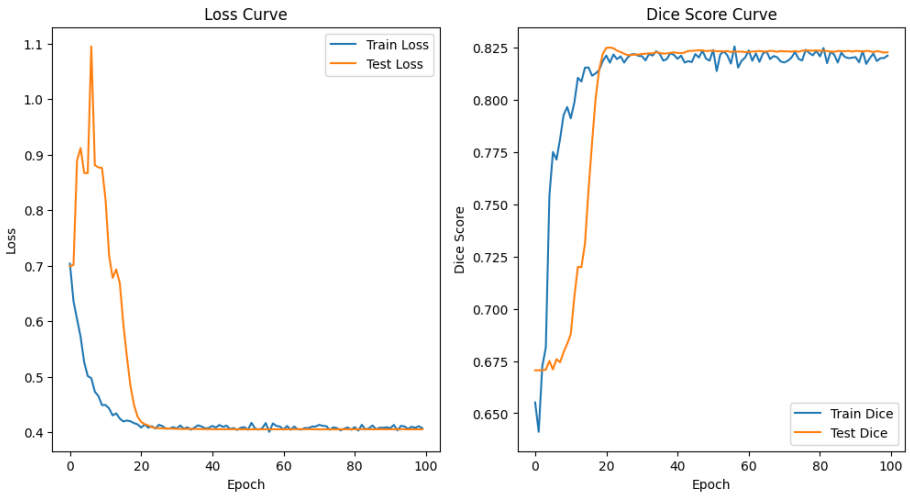


We can visually see by ourselves that most of the target areas have been missed.

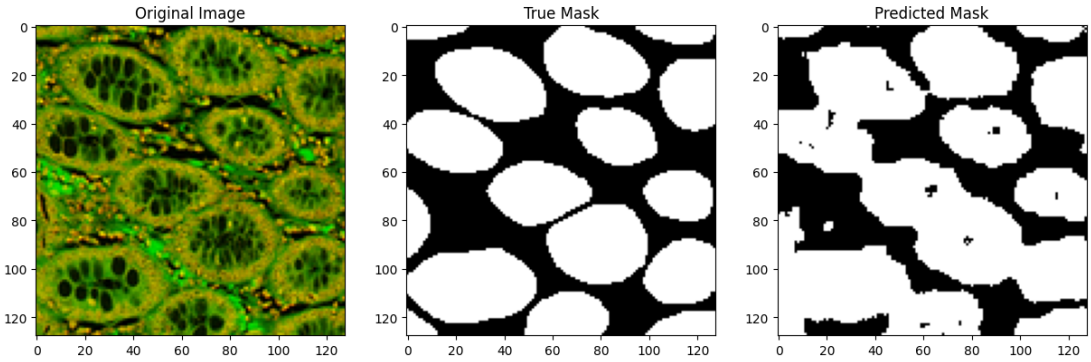
2.2 Exercise 2.2

2.2.1 Variation 1/3

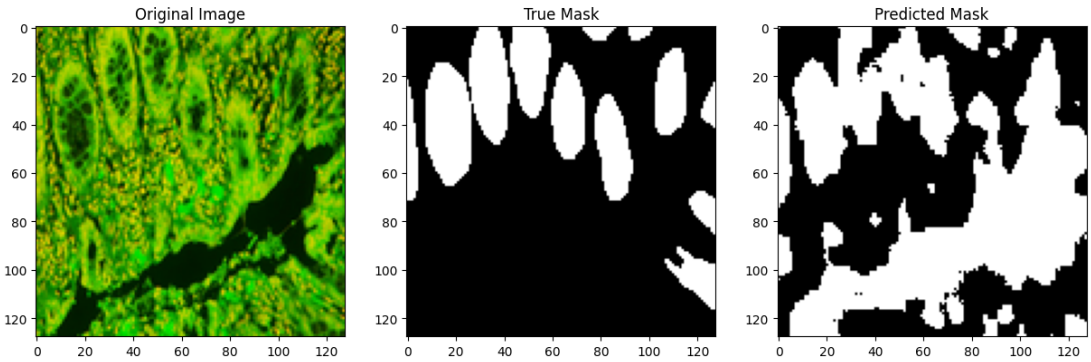
To increase performance we use batch normalization. We apply it after every convolutional and "deconvolutional" layer. Again, the model is trained for 100 epochs with a learning rate of 0.01 and a batch size of 32. We achieved a dice score of 0.82.



Lets inspect some cases:
Dice score: 0.91



Dice score: 0.54

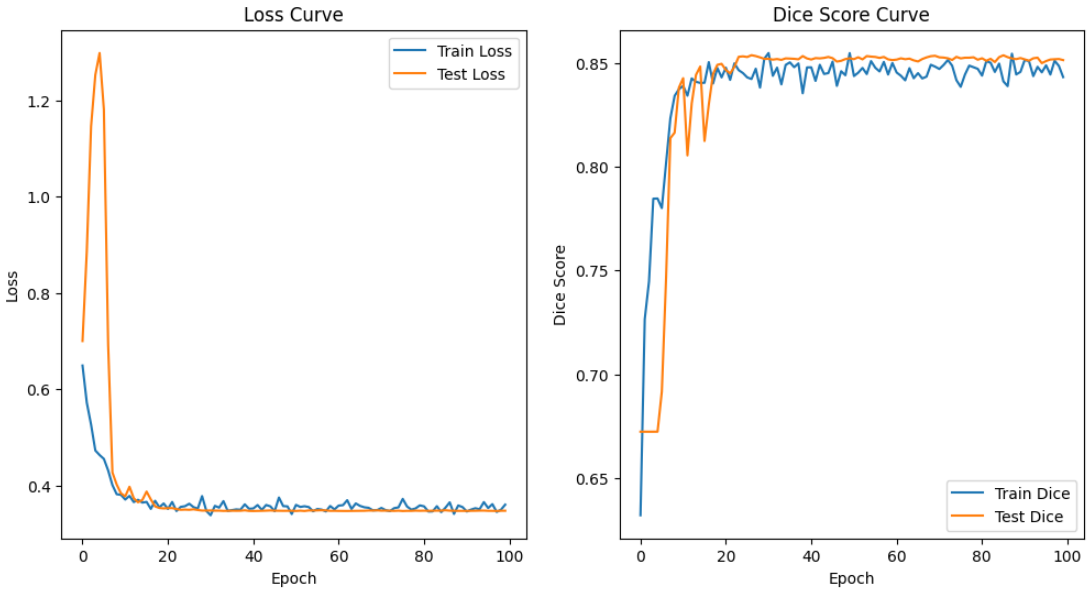


Comparing the two cases, it becomes clear why there is a huge difference in dice scores between them. In the first one the network managed to identify roughly where the cells are and their overall structure. In the second case however, the predicted mask has pointed out the negative space as cells.

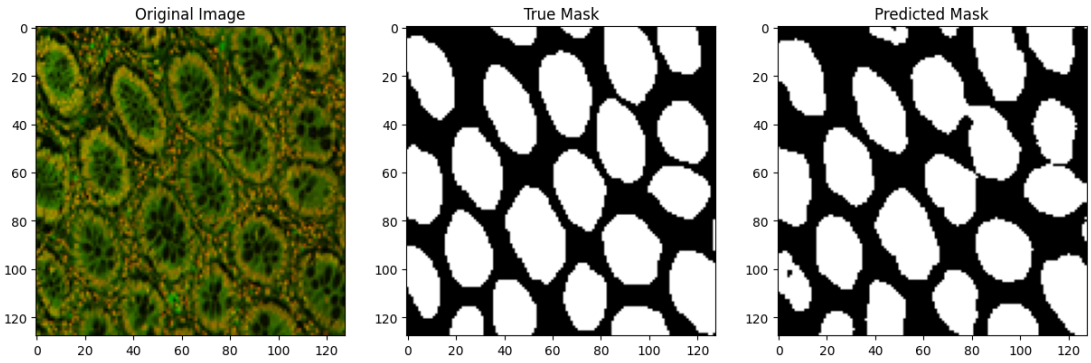
From these examples we can deduce that there is still room for improvement.

2.2.2 Variation 2/3

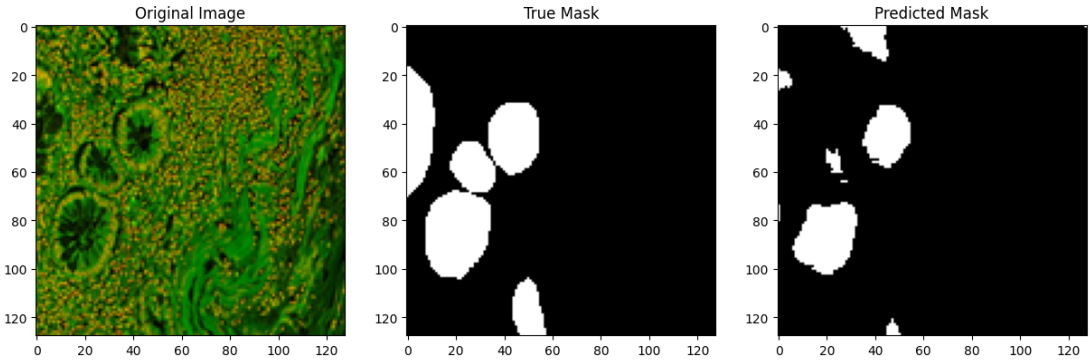
The train test is quite small so having a large batch size doesn't leave a lot of room for parameter adjustments. For that reason, we reduce the batch size to 16, keeping all other parameters the same. We achieved a dice score of 0.85.



Lets inspect some cases: Dice score: 0.92



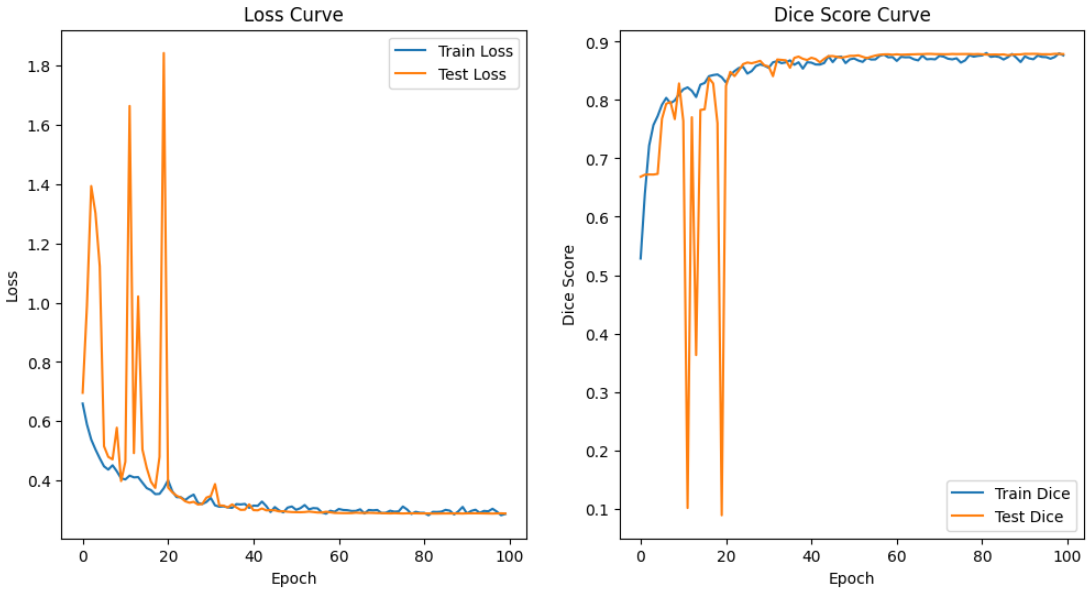
Dice score: 0.58



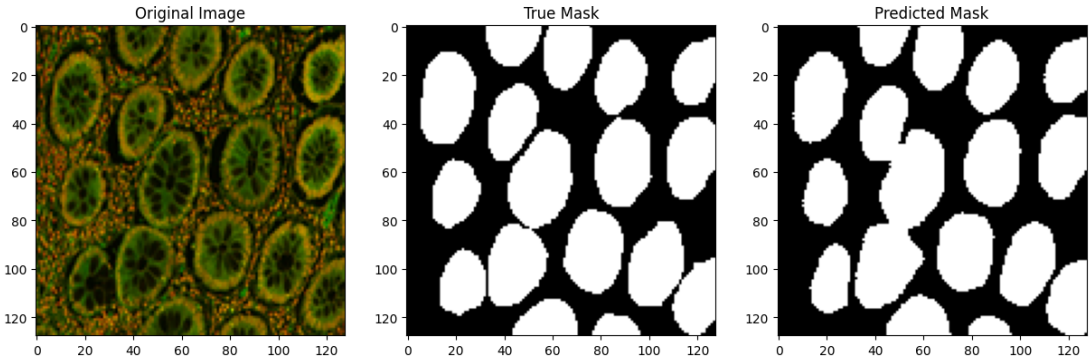
We have started to get some pretty good results.

2.2.3 Variation 3/3

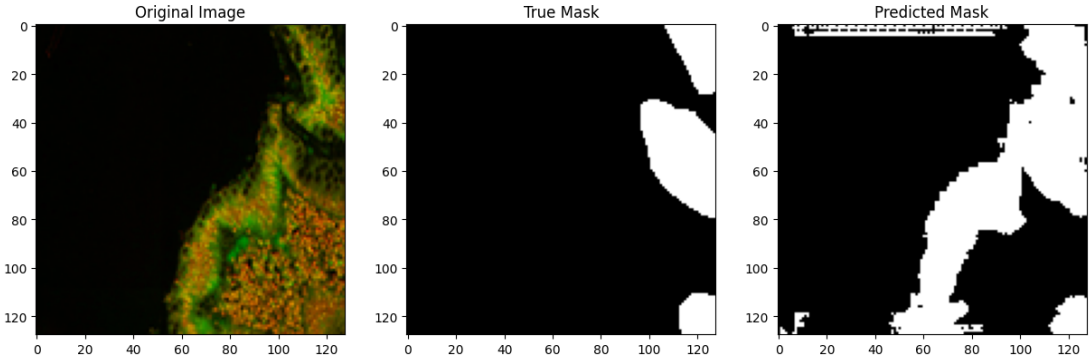
The previous model seemed to perform reasonably, so no changes were made in the architecture. Instead, we made changes to the learning rate scheduler of the optimizer. Previously it was a linear step optimizer, with gama=0.2 and step_size of 5. As a result, the learning rate is reduced way too quickly and not a lot of improvement can be done. To combat this, the step_size is set to 20. We achieved a dice score of 0.88.



Lets view some examples: Dice score: 0.95



Dice score: 0.53:



In the worse example, we can clearly see some checkboard patterns and some weird artifacts. I am not sure if it is from overfitting the model but not a lot of these cases exist. However we can see that in the dice score of 0.95, the predicted mask is almost perfect.