Deep Learning for Image Analysis

1MD120 – Report for Assignment 3

Elise Chin

May 5, 2021

1 Classification of handwritten digits using a Convolutional Neural Network

In this assignment task we will again perform classification of the MNIST dataset, this time using a convolutional neural network instead of the fully connected network used in assignment 2.

I used Google Colab to perform all experiments. Firstly, I imported the `MNIST.zip` file in my Google Drive, then extracted all the files into the current directory of the notebooks using the function `extract_files()`.

```
def extract_files(file_name):
    """Extract the input zip file in the current directory."""

    with zipfile.ZipFile(file_name, 'r') as zip:
        zip.extractall()
```

We keep the original shape of the images and we do not one hot encode the labels. Training and test loaders are then created with `torch.utils.data.DataLoader`.

In the following, training losses and accuracies are measured on the current mini-batch and averaged to get the training loss and accuracy for an epoch. Test loss and accuracy are computed on the whole test data for each epoch.

Exercise 1.1: FNN

> *Start by implementing exactly the same (fully connected) network as you designed for Assignment 2 using built-in PyTorch functions. Try to set learning parameters to imitate your code from Assignment 2; it is fine to use the default weight initialization scheme of the framework. Compare the performance with your Assignment 2 results; do you observe similar accuracy? How many times faster/slower was your own implementation?*

In Assignment 2, my best model has an input layer with 784 (28×28) neurons, one hidden layer with 800 neurons, and an output layer with 10 neurons, one for each class. ReLU was used as the activation function. The following code corresponds to the same network using built-in PyTorch functions.

```
class FNN(nn.Module):
```

```
    def __init__(self, input_size=784, hidden_size=800, num_classes=10):
      super(FNN, self).__init__()
      self.fc1 = nn.Linear(input_size, hidden_size)
      self.relu = nn.ReLU()
      self.fc2 = nn.Linear(hidden_size, num_classes)

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

Note that we do not add a softmax at the end of the network since we choose `nn.CrossEntropyLoss()` as the loss criterion which combines `nn.LogSoftmax()` and `nn.NLLLoss()` according to the documentation.

Similarly to the experiment in the previous assignment, the network is trained for 30 epochs using Stochastic Gradient Descent with a learning rate of 1 and a mini-batch size of 1024. We observe similar accuracy: both models were able to reach an accuracy of **98%**. However, this implementation was faster than the previous one. It took only 50 seconds to train the model with Google Colab's GPU, whereas it took me almost an hour when I trained my model on my own computer. This can be explained by the implementation choice and by the difference in computers' specifications (mine is *Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz* and 8GB of RAM). Below we plot the model's loss and accuracy over 30 epochs.

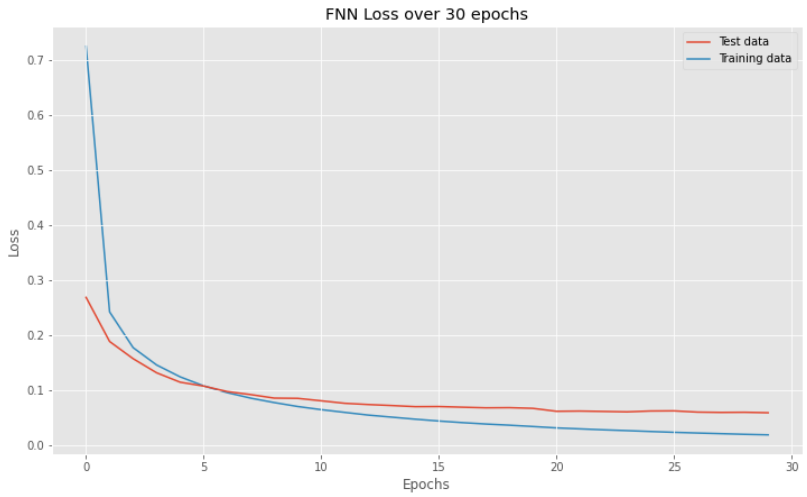


Figure 1: FNN Loss on training and test data vs. number of epochs. Trained with stochastic gradient descent with a learning rate of 1 and a mini-batch size of 1024, the model is able to converge quickly. Both curves seem to continue to decrease, suggesting that the model can be trained longer. One can observe that the training loss decreases faster than the test loss, but the gap is very negligible (0.04 difference)

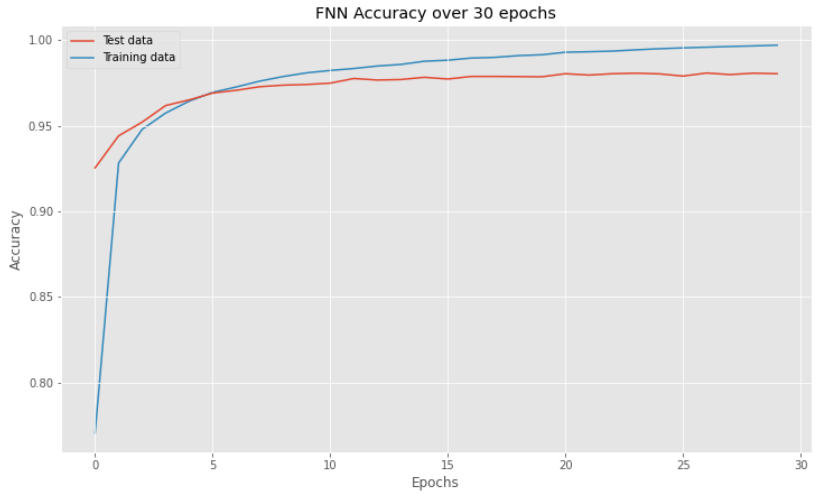


Figure 2: FNN Accuracy on training and test data vs. number of epochs. Trained with stochastic gradient descent with a learning rate of 1 and a mini-batch size of 1024, the model is able to reach an accuracy over 98% on test data and an accuracy of 99.7% on training data at the end of 30^{th} epoch. If we have trained longer, the model might reach a perfect accuracy on the training data, but then it would be a case of overfitting.

By observing the slight difference between the training and test accuracy curves, we can say that the network is overfitting the training data. We could have stopped the learning after 5 epochs, which corresponds to the moment when the curves intersect.

Exercise 1.2: CNN

Let us now exchange the Fully connected architecture for a Convolution based one. Construct the network specified below in PyTorch. Train the network using standard Stochastic Gradient Descent (SGD) on the MNIST dataset. Choose a suitable mini-batch size and number of training epochs (observing a reasonable convergence) and a learning rate which gives you a good convergence without waiting forever. You are expected to reach above 98% accuracy on the test data partition. How many learnable weights does this network contain? Compare with how many weights you had in the previous exercise.

We build a Convolutional Neural Network (CNN) based on the following architecture:

1)	Convolution	8 times 3x3x1 convolutions with stride 1 and padding 1
2)	ReLU	Non-linearity
3)	Max Pooling	2x2 max pooling with stride 2
4)	Convolution	16 times 3x3x8 convolutions with stride 1 and padding 1
5)	ReLU	Non-linearity
6)	Max Pooling	2x2 max pooling with stride 2
7)	Convolution	32 times 3x3x16 convolutions with stride 1 and padding 1
8)	ReLU	Non-linearity
9)	Fully Connected	10 fully connected layer
10)	Softmax	Softmax layer
11)	Classification Output	Crossentropy with 10 classes

```python
class CNN(nn.Module):
    def __init__(self):
      super(CNN, self).__init__()

      self.model = nn.Sequential(
          nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, stride=1,
          ↪  padding=1), # output: 8x28x28 (c,w,h)
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2), # output: 8x14x14
          nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1), # output: 16x14x14
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2), # output: 16x7x7
          nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1), # output: 32x7x7
          nn.ReLU()
      )

      self.classifier = nn.Sequential(
          nn.Linear(32*7*7, 10)
      )

    def forward(self, x):
      x = self.model(x)
      x = self.classifier(x.view(x.size(0), -1))
      return x
```

Again, we do not add a softmax layer because it will be applied when computing the cross-entropy loss.

This time, the network is trained for 20 epochs using Stochastic Gradient Descent with a learning rate of 0.1 and a mini-batch size of 32. We obtained an accuracy of **99%** on the test data partition within 2 minutes and 40 seconds. The CNN performs better than FNN. Below we plot the loss and accuracy on both training and test data.

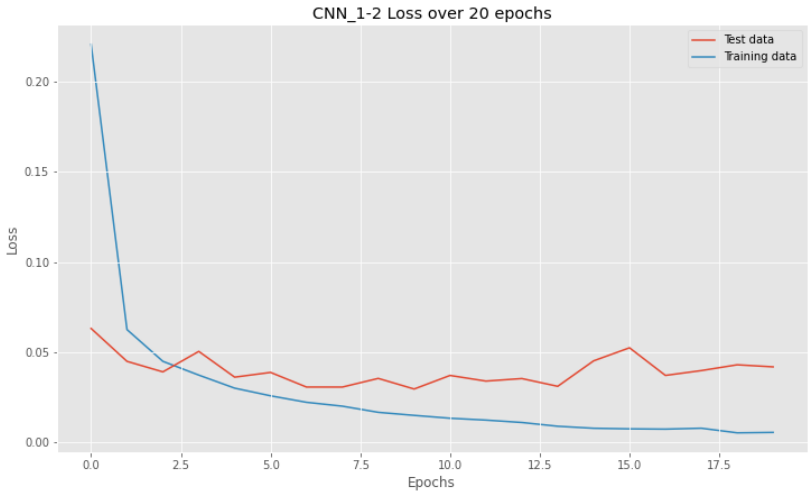


Figure 3: CNN Loss on training and test data vs. number of epochs. Trained with stochastic gradient descent with a learning rate of 0.1 and a mini-batch size of 32, the model is able to converge quickly. It is less stable than the FNN, but it gives better results. The difference between the two curves is also smaller than the one obtained with FNN (0.03 vs. 0.04).

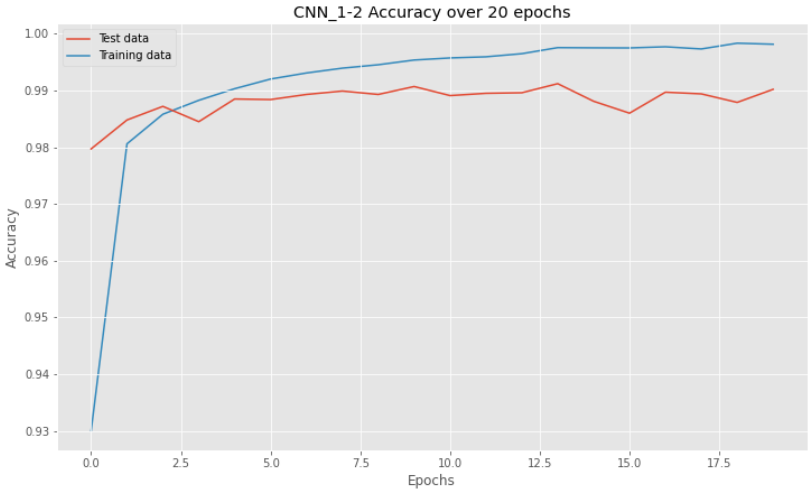


Figure 4: CNN Accuracy on training and test data vs. number of epochs. Trained with stochastic gradient descent with a learning rate of 0.1 and a mini-batch size of 32, the model is able to reach an accuracy over 99% on test data and an accuracy of 99.8% on training data at the end of 30^{th} epoch. The model performs better than the FNN and less overfits.

The CNN still overfits a little but less than the FNN. We only have a difference of 0.008 between the training and test accuracy.

Exercise 1.3: Swap ReLU and Pooling layers

> *In the previous exercise we first applied the activation function (ReLU in this case) and then performed Max pooling. What would change (accuracy, speed) if you swap the order of these two operations? Test your hypothesis empirically by exchanging layers 2 and 3, resp. layers 5 and 6, such that the pooling operation is performed before the ReLU. The difference is probably more distinct for the hyperbolic tangent activation function, why?*

Firstly, we swap the pooling and ReLU layers so that each convolution layer is directly followed by a Max pooling layer.

```
class CNN_POOL_RELU(nn.Module):
    def __init__(self):
      super(CNN_POOL_RELU, self).__init__()

      self.model = nn.Sequential(
          nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, stride=1,
          ↪  padding=1), # output: 8x28x28 (c,w,h)
          nn.MaxPool2d(kernel_size=2, stride=2), # output: 8x14x14
          nn.ReLU(),
          nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1), # output: 16x14x14
          nn.MaxPool2d(kernel_size=2, stride=2), # output: 16x7x7
          nn.ReLU(),
          nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1), # output: 32x7x7
          nn.ReLU()
```

```
        )

        self.classifier = nn.Sequential(
            nn.Linear(32*7*7, 10)
        )

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

We trained the network with the same parameters as before (20 epochs, learning rate of 0.1, mini-batch size of 32). We were able to reach an accuracy of **99%**, which is similar to the previous CNN, within the same amount of time (2 minutes and 40 seconds). The curves for both training and test losses and accuracies also look similar. So it seems that swapping the pooling operation and the ReLU layer does not impact neither the speed or the network's accuracy. The accuracy did not change, probably because ReLU and MaxPool are both Max function, so we can commute the two functions safely. However, we could expect an improvement in speed, since ReLU is applied on less inputs because the Max pooling layer reduces the number of parameters beforehand. This optimization is negligible for the CNN, because majority of the time is used in convolutional layers.

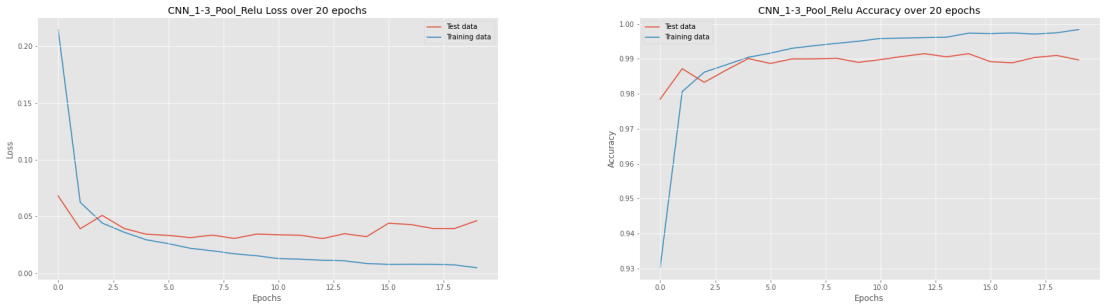


Figure 5: Performance of CNN with pooling and ReLU layers swapped on training and test data. We still reach an accuracy of 98%, similar results as the original CNN. The training and test curves for both loss and accuracy also look the same.

To see if there is any difference that we should observe, we also replace the ReLU activation function with Tanh.

```
class CNN_POOL_TANH(nn.Module):
    def __init__(self):
        super(CNN_POOL_TANH, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, stride=1,
            ↪  padding=1), # output: 8x28x28 (c,w,h)
            nn.MaxPool2d(kernel_size=2, stride=2), # output: 8x14x14
            nn.Tanh(),
            nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1), # output: 16x14x14
            nn.MaxPool2d(kernel_size=2, stride=2), # output: 16x7x7
            nn.Tanh(),
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1), # output: 32x7x7
            nn.Tanh()
        )

        self.classifier = nn.Sequential(
```

```
            nn.Linear(32*7*7, 10)
        )

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

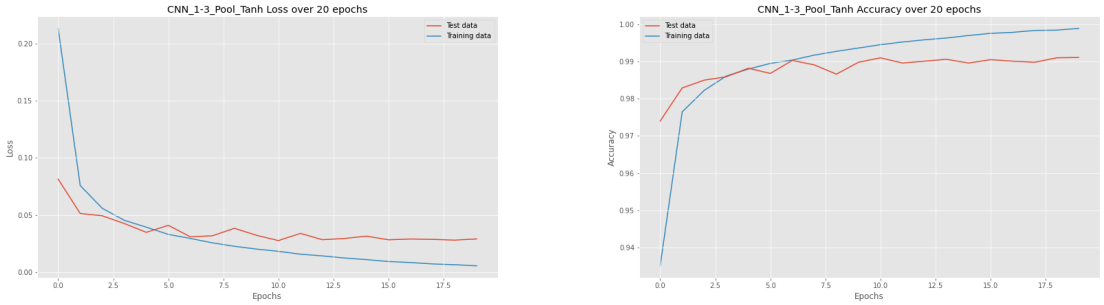


Figure 6: Performance of CNN with pooling and Tanh layers swapped on training and test data. We still reach an accuracy of 98%, similar results as the original CNN. The training and test curves for both loss and accuracy also look the same.

When replacing ReLU activation function with Tanh, we were also able to reach an accuracy of **99%**, which is similar to the previous CNN, within the same amount of time (2 minutes and 40 seconds). The curves for both training and test losses and accuracies also look similar. So it seems that swapping the pooling operation and the Tanh layer does not impact neither the speed or the network's accuracy. Even though the learnt function is different, we obtain the same result maybe because the CNN and the task are not that complex.

Exercise 1.4: Adam optimizer

> *Now change the optimizer, training the network with ADAM instead of SGD. It is fine to pick the default parameters proposed by PyTorch. (Commonly appearing default values are: GradientDecayFactor (β_1): 0.9000, SquaredGradientDecayFactor (β_2): 0.9990, ϵ: 10^{-8}.) Do you manage to get better results faster than when using the plain SGD optimization?*

We used the same network as in Exercise 1.2 (1), but this time we change the optimizer to Adam. We also choose the default learning rate 0.001. We manage to get better results faster than when using the plain SGD optimization. We reach an accuracy of **99%** after the second epoch using Adam, whereas we get the same result after 5 epochs using SGD optimization. Adam uses Momentum and Adaptive Learning Rates to converge faster. Momentum helps us to progress faster by adding some fraction of the previous update to the current update, so that repeated updates in a particular direction compound. We build up momentum, moving faster and faster in the desired direction. With adaptive learning rates, we start off with big steps and finish with small steps. We are then allowed to move faster initially. As the learning rate decays, we take smaller and smaller steps, allowing us to converge faster, since we don't overstep the local minimum with as big steps.

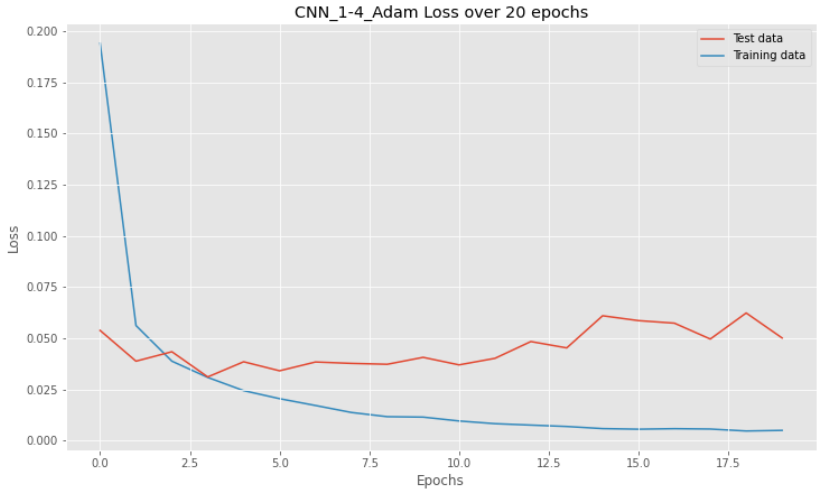


Figure 7: CNN Loss on training and test data vs. number of epochs. Trained with Adam, a learning rate of 0.001 and a mini-batch size of 32, the model is able to converge faster. At the end of the 20^{th} epoch, we reach a loss of 0.005.

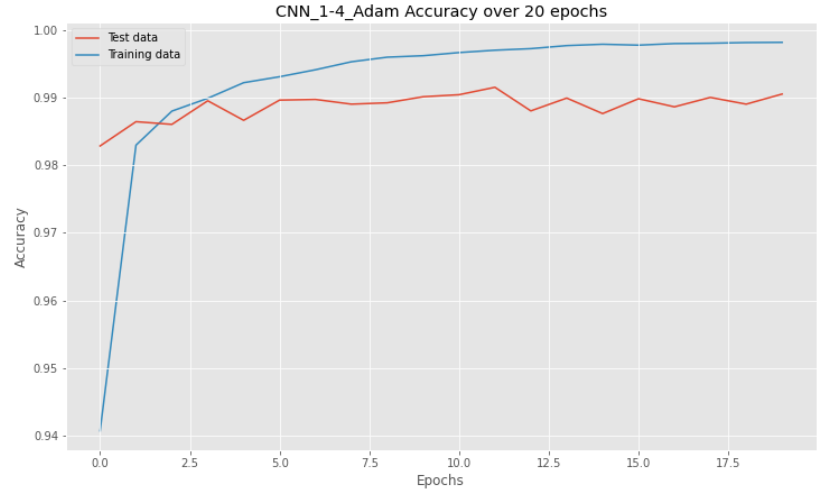


Figure 8: CNN Accuracy on training and test data vs. number of epochs. Trained with Adam, a learning rate of 0.001 and a mini-batch size of 32, the model is able to reach an accuracy over 99% on test data faster than using plain SGD optimization (5 epochs vs 7 epochs). We still observe the same kind of overfitting.

Exercise 1.5

Try a few (at least 3) variations based on what you have learned in the course so far; this could be architectural changes, various regularization approaches, change of optimization method, learning-rate scheme, change of activation function, etc. How good performance do you manage to reach?

Varying the number of convolution blocks.

For the first experiment, we try to vary the number of Convolution-Activation-Pooling block. In order to do that, our `CNN_block` class takes as an argument the number of desired blocks `num_blocks` between 1 and 4. Depending on this parameter, we stack `num_blocks - 1` convolution (C), ReLU (R), and Max Pooling (P) layers, followed by one convolution and ReLU layers. We also compute the output size for each case, which will be the input of the last `Linear` layer. For 4 blocks, the last Max pooling layer has a stride of 1 because we do not want to reduce the number of parameters too much to apply a convolution layer after.

```python
class CNN_block(nn.Module):
    def __init__(self, num_blocks):
      super(CNN_block, self).__init__()

      self.conv1 = nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, stride=1, padding=1) # output: 8x28x28 (c,w,h)
      self.conv2 = nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1) # output after pool2: 16x14x14
      self.conv3 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1) # output after pool2: 32x7x7
      self.conv4 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1) # output after pool1: 64x6x6

      self.relu = nn.ReLU()
      self.pool1 = nn.MaxPool2d(kernel_size=2, stride=1)
      self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

      if num_blocks == 1:
        self.model = nn.Sequential(
            self.conv1,
            self.relu
        )
        output_size = 8*28*28
      elif num_blocks == 2:
        self.model = nn.Sequential(
            self.conv1,
            self.relu,
            self.pool2,
            self.conv2,
            self.relu
        )
        output_size = 16*14*14
      elif num_blocks == 3:
        self.model = nn.Sequential(
            self.conv1,
            self.relu,
            self.pool2,
            self.conv2,
            self.relu,
            self.pool2,
            self.conv3,
            self.relu
        )
        output_size = 32*7*7
      elif num_blocks == 4:
        self.model = nn.Sequential(
            self.conv1,
            self.relu,
            self.pool2,
```

```
            self.conv2,
            self.relu,
            self.pool2,
            self.conv3,
            self.relu,
            self.pool1,
            self.conv4,
            self.relu
        )
        output_size = 64*6*6
      else:
        raise ValueError("Number of convolution layers between 1 and 4.")

      self.classifier = nn.Sequential(
          nn.Linear(output_size, 10)
      )

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

All models were trained for 20 epochs using Adam optimizer since we reach a faster convergence with it. We also use the default learning rate 0.001, set Gradient-DecayFactor $\beta_1 = 0.9000$, SquaredGradientDecayFactor $\beta_2 = 0.9990$, $\epsilon = 10^{-8}$, and use a mini-batch size of 32 as usual. On the following plot, we see that the two best performing models are the one with 2 and 3 convolutional blocks, represented by the blue and purple curves respectively.

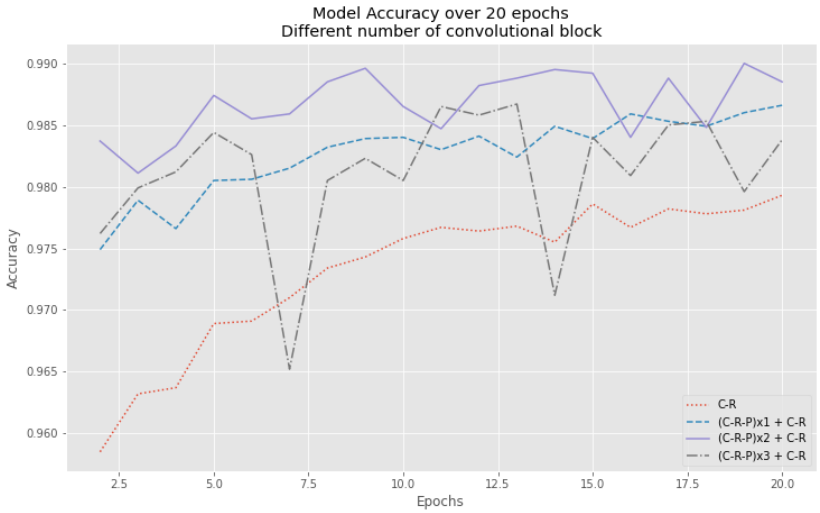


Figure 9: CNN Accuracy on test data vs. number of epochs. We do not represent the first epoch to focus on the later epochs. We can see that the two best performing models are the one with 2 and 3 convolutional blocks.

We now represent their test accuracies along with their training accuracies to see which one overfit the less.

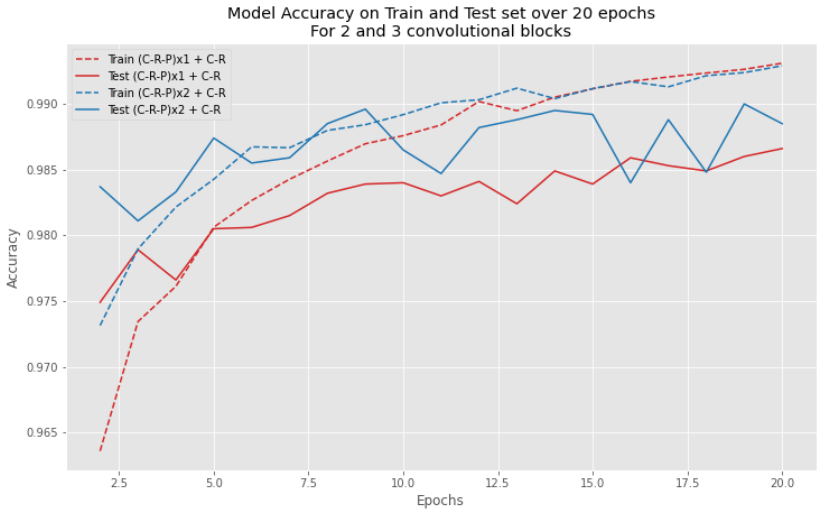


Figure 10: Accuracy on training and test data of the CNNs with 2 and 3 convolutional blocks vs. number of epochs. We do not represent the first epoch to focus on the later epochs. The gap between the blue curves (3 blocks) is smaller than the gap between the red curves (2 blocks), so the CNN with 3 convolutional blocks less overfit.

From the results above, it is best to choose the model with 3 convolutional blocks as it gives better accuracy and the model is less overfitting than the other one. More layers do not necessarily entail better performance, and this is probably because our dataset is rather small so we do not need to have very deep neural network to obtain reasonable performance.

Varying the number of feature maps.

Another parameter that can be chosen in a CNN architecture is the number of features maps in each convolution layers. We try several number of feature maps in the first layer (`[8, 16, 32, 64]`). For the next layers, we double the number of the previous one. So for example, if we have 8 feature maps in the first convolution layer, then in the second we will have 16, and in the third 32.

```
class CNN_feature_maps(nn.Module):
    def __init__(self, num_feature_maps):
      super(CNN_feature_maps, self).__init__()
      self.num_feature_maps = num_feature_maps
      self.model = nn.Sequential(
          nn.Conv2d(in_channels=1, out_channels=num_feature_maps, kernel_size=3,
          ↪  stride=1, padding=1),
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2),
          nn.Conv2d(num_feature_maps, num_feature_maps*2, kernel_size=3, stride=1,
          ↪  padding=1),
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2),
          nn.Conv2d(num_feature_maps*2, num_feature_maps*4, kernel_size=3, stride=1,
          ↪  padding=1),
          nn.ReLU()
      )

      self.classifier = nn.Sequential(
          nn.Linear(num_feature_maps*4*7*7, 10)
      )
```

```
19
20      def forward(self, x):
21        ...
```

All models were trained for 20 epochs using Adam optimizer since we reach a faster convergence with it. We also use the default values and a mini-batch size of 32 as usual. On the following plot, we see that the two best performing models are the one with 16 and 32 feature maps in the first layer.

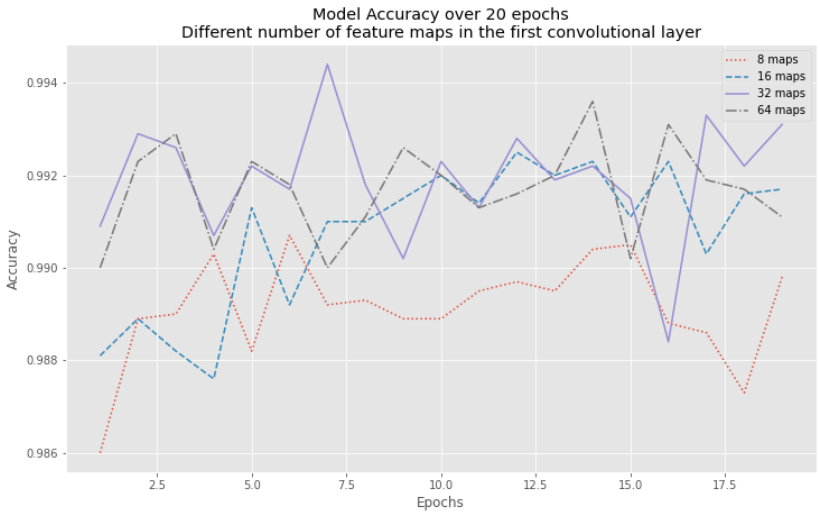


Figure 11: CNN Accuracy on test data vs. number of epochs. We do not represent the first epoch to focus on the later epochs, which also explains the fluctuation. We can see that the two best performing models are the one with 16 and 32 feature maps in the first layer.

We now represent their test accuracies along with their training accuracies to see which one overfit the less.

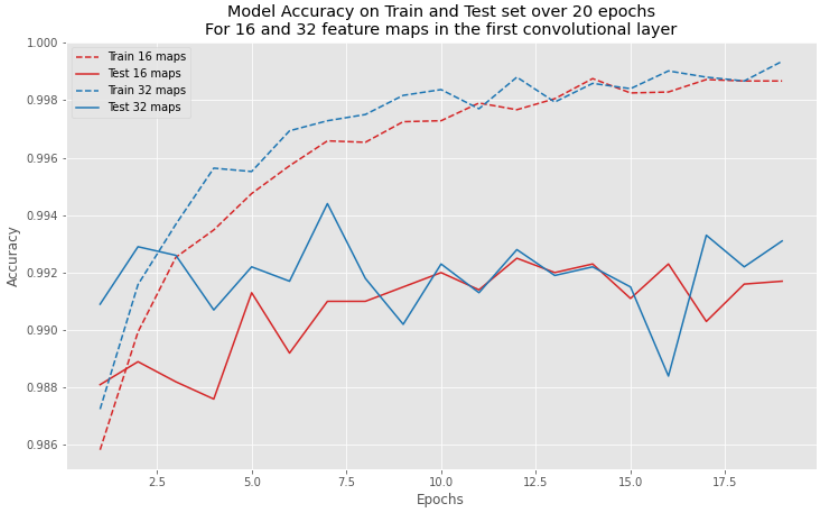


Figure 12: Accuracy on training and test data of the CNNs with 16 and 32 feature maps in the first convolution layer vs. number of epochs. We do not represent the first epoch to focus on the later epochs. The gap between the blue curves (32 feature maps) is smaller than the gap between the red curves (16 feature maps), so the CNN with 32 feature maps in the first convolution layer less overfit.

From the results above, it is best to choose the model with 32 feature maps in the first convolution layer as it gives better accuracy and the model is less overfitting than the other one. Digits caracteristics seem to be better learnt with more feature maps.

Activation function choice (ReLU vs Sigmoid vs Tanh).

As we saw in Exercise 1.3 (1), replacing ReLU by Tanh does not impact the model's performance. We also want to see if using another activation function such as sigmoid will change anything. For that, our class `CNN_act` takes as a parameter the name of the desired activation function and applies it after every convolution layer and before max pooling layers. Note that we use 32 feature maps in the first convolution layer as we see that it gave the best performance among all the other number of feature maps tested.

```
class CNN_act(nn.Module):
    def __init__(self, activation):
      super(CNN_act, self).__init__()

      if activation == "relu":
        self.activation = nn.ReLU()
      elif activation == "sigmoid":
        self.activation = nn.Sigmoid()
      elif activation == "tanh":
        self.activation = nn.Tanh()

      self.model = nn.Sequential(
          nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1,
          ↪  padding=1),
          self.activation,
          nn.MaxPool2d(kernel_size=2, stride=2),
          nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
          self.activation,
          nn.MaxPool2d(kernel_size=2, stride=2),
          nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
          self.activation
      )

      self.classifier = nn.Sequential(
          nn.Linear(128*7*7, 10)
      )

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

All models were trained for 20 epochs using Adam optimizer since we reach a faster convergence with it. We also use the default values of PyTorch for Adam and a mini-batch size of 32 as usual. On the following plot, we can see that using sigmoid as the activation function makes the CNN reach an accuracy above 95% slower than the other models. Hence, there is no advantage in using sigmoid and we will continue using ReLU.

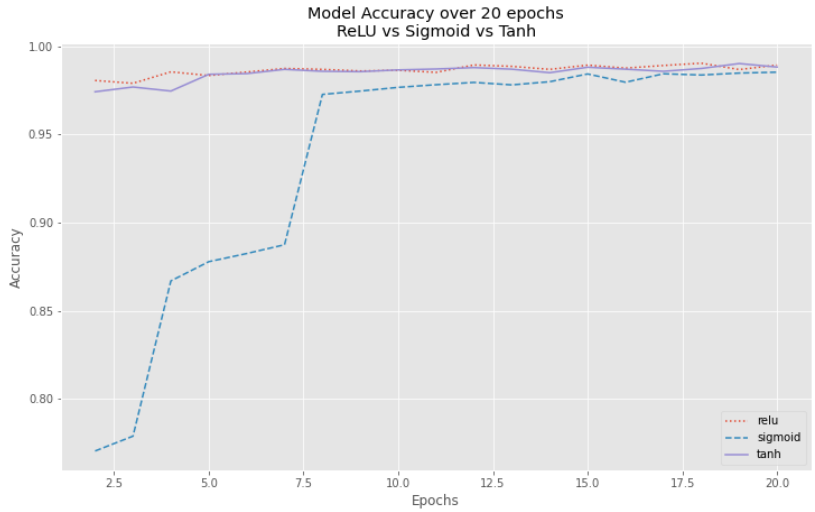


Figure 13: CNN Accuracy on test data vs. number of epochs. We do not represent the first epoch to focus on the later epochs, which also explains the fluctuation. The model using sigmoid as the activation function takes more time to reach the same accuracy as the other models.

Dropout.

Dropout is a simple regularization technique to prevent neural networks from overfitting by randomly selecting neurons to ignore during training. This means that those neurons do not contribute in the forward pass and weight updates are not applied during the backward pass. The other neurons will thus handle the inputs' representation of the missing neurons to make good predictions. The network becomes less sensitive to the specific weights of neurons, which results in a network that is capable of better generalization and is less likely to overfit the training data.

In PyTorch, dropout is implemented using the `Dropout2d()` method with a given probability p. In this section, we experiment different dropout rate: 0%, 10%, 20%, 30%, 40%, 50%, or 60%. Dropout layers are added after each Max pooling layers.

```
class CNN_dropout(nn.Module):
    def __init__(self, dropout):
      super(CNN_dropout, self).__init__()
      self.model = nn.Sequential(
          nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1,
          ↪  padding=1),
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2),
          nn.Dropout2d(p=dropout),
          nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2),
          nn.Dropout2d(p=dropout),
          nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
          nn.ReLU()
      )

      self.classifier = nn.Sequential(
          nn.Linear(128*7*7, 10)
      )

```

```
21      def forward(self, x):
22          ...
```

All models were trained for 20 epochs using Adam optimizer since we reach a faster convergence with it. We also use the default learning rate and a mini-batch size of 32 as usual. On the following plot, we see that the two best performing models are the one with no dropout and a dropout of 10%. Dropout may help the network to generalize the data, but in our case, it does not lead to a better accuracy performance.

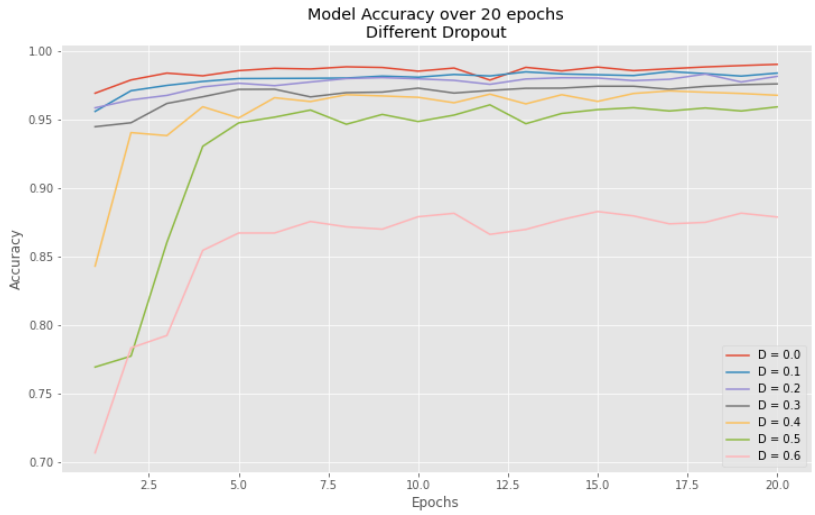


Figure 14: CNN Accuracy on test data vs. number of epochs. We do not represent the first epoch to focus on the later epochs. We can see that the two best performing models are the one with no dropout and a dropout of 10%

We now represent their test accuracies along with their training accuracies to see which one overfit the less.

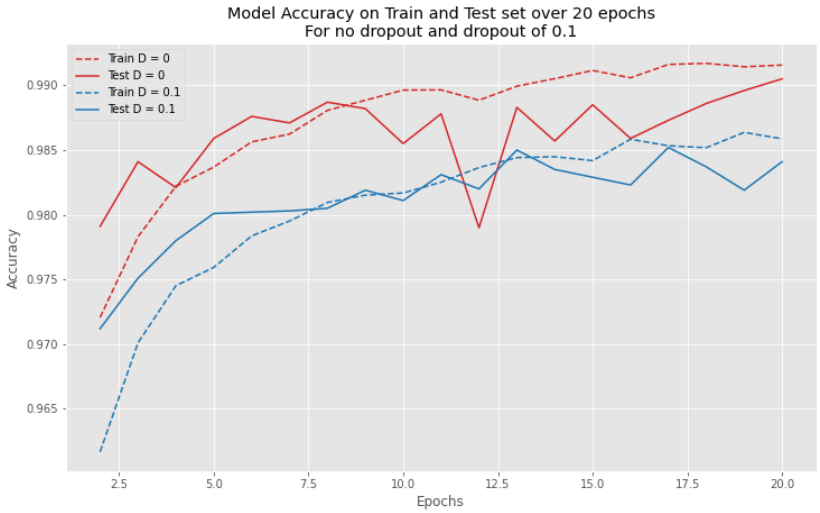


Figure 15: Accuracy on training and test data of the CNNs with no dropout and a 10% dropout vs. number of epochs. We do not represent the first epoch to focus on the later epochs. No dropout leads to better accuracy.

From the results above, it is best to choose the model with no dropout because it gives better accuracy. However, it is interesting to note that the model with a 10% dropout less overfits the training data, compared to the model with no dropout. So dropout indeed helps to improve the generalization of the neural network to new data.

Batch Normalization.

Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch. It makes deep networks much easier to train, improves gradient flow and enables faster convergence. Batch normalization layers are usually inserted after convolutional layers and before nonlinearity. This is what we do in the `CNN_bn` class. The number of features matches the number of feature maps in the preceding convolution layer.

```
class CNN_bn(nn.Module):
    def __init__(self):
      super(CNN_bn, self).__init__()
      self.model = nn.Sequential(
          nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1,
          ↪  padding=1),
          nn.BatchNorm2d(32),
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2),
          nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
          nn.BatchNorm2d(64),
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2),
          nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
          nn.BatchNorm2d(128),
          nn.ReLU()
      )

      self.classifier = nn.Sequential(
          nn.Linear(128*7*7, 10)
      )

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

We compare this model's performance with the CNN without batch normalization. The results are very satisfying.

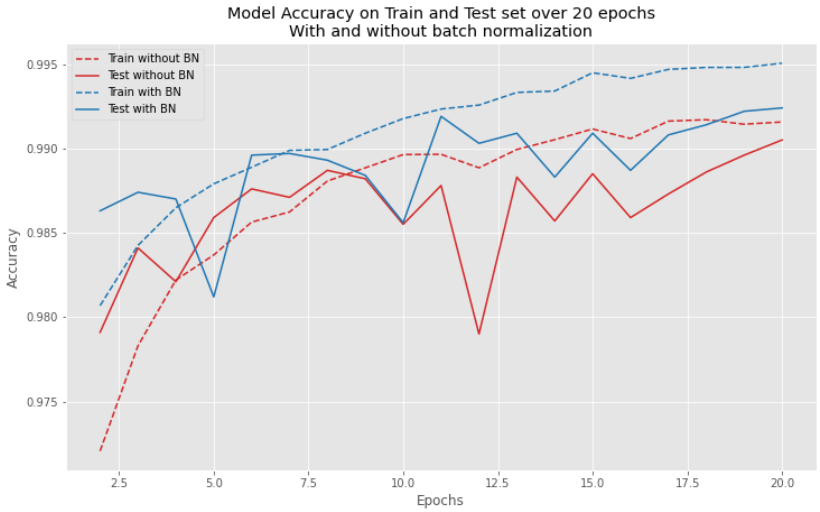


Figure 16: Accuracy on training and test data of the CNNs with batch normalization and without vs. number of epochs. We do not represent the first epoch to focus on the later epochs. The CNN with batch normalization after each convolution layer performs better and does not overfit that much (0.003 point difference).

L1 and L2 Regularization with SGD.

L1 and L2 regularizations introduce a penalty to the loss function when training a neural network to encourage the network to use small weights. And with smaller weights, a model can be more stable and less likely to overfit the training dataset, in turn having better performance when making a prediction on new data.

For L1 regularization, we need to implement it manually as the optimizer in PyTorch only implements L2 regularization. So in our `train_model()` function, we add the penalty to the loss before the back-propagation. Here we set $\lambda = 0.001$.

```
def train_model(model, loss_criterion, optimizer, train_loader, X_test, Y_test,
    num_epochs, is_L1=False, plot_result=False):
  ...

  for epoch in range(num_epochs):
      ...

      for i_b, batch in enumerate(train_loader):
          x_train, y_train = batch[0].to(device), batch[1].to(device)

          model.zero_grad()

          # 1. Forward propagation
          output = model(x_train)

          # 2. Compute loss and accuracy on current mini-batch
          loss = loss_criterion(output, y_train)
          ...

          # 3. Backpropagation
          if is_L1:
            regularization_term = 0
```

```
            for param in model.parameters():
                regularization_term += torch.sum(abs(param))

            loss = loss + 0.001 * regularization_term

        loss.backward()

        # 4. Update parameters
        optimizer.step()

    ...
```

For L2 regularization, we set the parameter `weight_decay` of the SGD optimizer to 0.001, same value as the λ for L1 regularization. Below, we plot the accuracies of the CNN with batch normalization with various loss regularizations and optimizers.

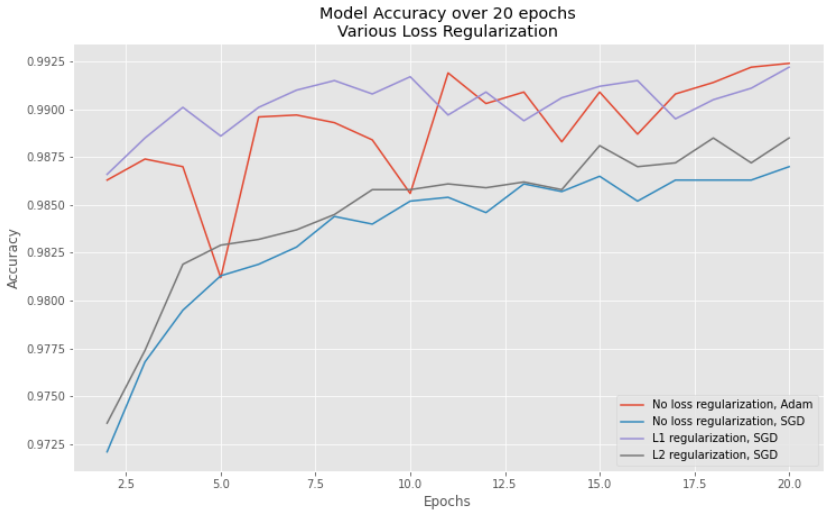


Figure 17: Accuracy on test data of CNNs with various loss regularization vs. number of epochs. We do not represent the first epoch to focus on the later epochs. The models with no loss regularization and Adam optimizer, and L1 optimization using SGD converge faster and give better results.

Based on our experiments, it seems that the models with no loss regularization and Adam optimizer, and L1 optimization using SGD converge faster and give better results.

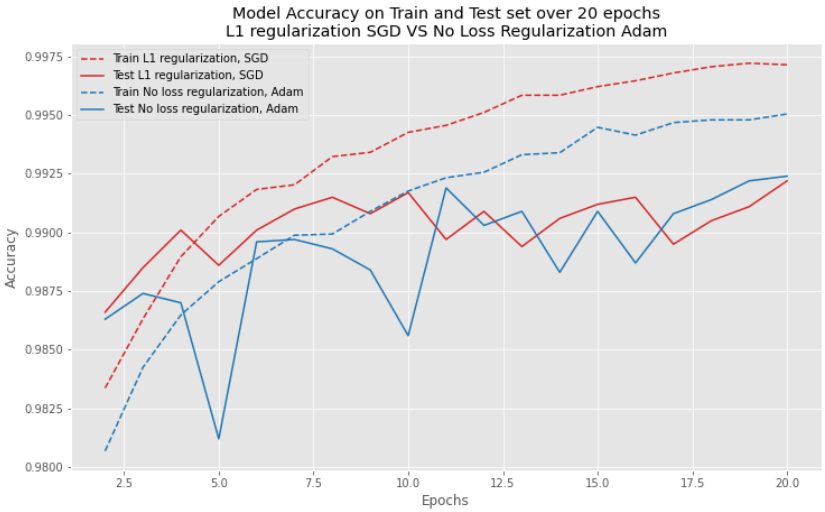


Figure 18: Accuracy on training and test data of the CNNs with L1 regularization using SGD and no loss regularization and Adam optimizer vs. number of epochs. We do not represent the first epoch to focus on the later epochs. CNN with Adam optimizer and no loss regularization still performs the best and less overfit.

CNN with batch normalization, Adam optimizer and no loss regularization still performs the best and less overfits.

Data Augmentation.

Finally, we try to augment the data to further improve our model by feeding it more samples. We choose to apply a rotation of 20 degrees to each training example. Hence, we double the size of the original dataset to have 120,000 images in total. We train for 20 epochs the CNN with batch normalization because it is the best model until now. Unexpectedly, we do not get a better performance with data augmentation. For the same number of epochs, we could reach an accuracy of **96%** with the augmented data, whereas with the original dataset, we reached an accuracy of **99%**.

Conclusion.

In conclusion, our best performing model is the CNN with 3 convolution-activation-pooling blocks, 32 feature maps in the first convolution layer, ReLU as activation function, no dropout, with batch normalization, trained with Adam optimizer and a learning rate of 0.001. The final accuracy on test data is **99.0%**.

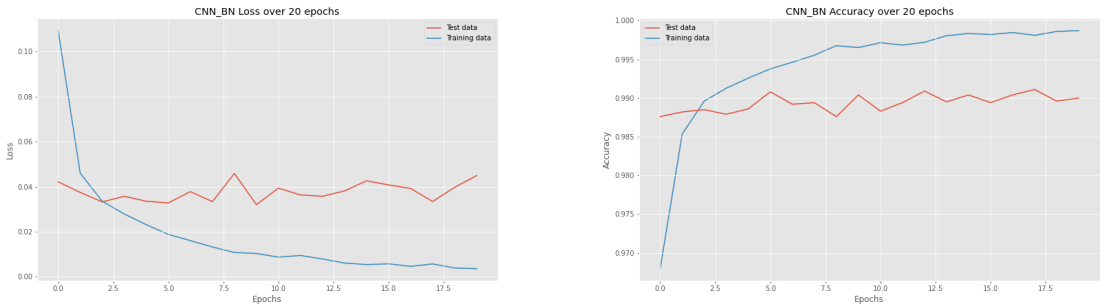


Figure 19: Best performing model loss and accuracy over 20 epochs. While the training loss curve seems to continue to decrease, the test loss curve seems to oscillate around 0.04. We observe the same trend in the accuracy plot. The model begins to overfit from the second epoch, but the difference is very negligible because we have already reached a good accuracy.

We also plot the resulting confusion matrix after evaluating the model on the test data partition. Since we have an accuracy of 99%, we can see on the confusion matrix an almost perfect classification per digit. There are a few misclassified images: for example, 10 images of the digit 9 were classified as 5, or 5 images representing the digit 3 were classified as 5 again.

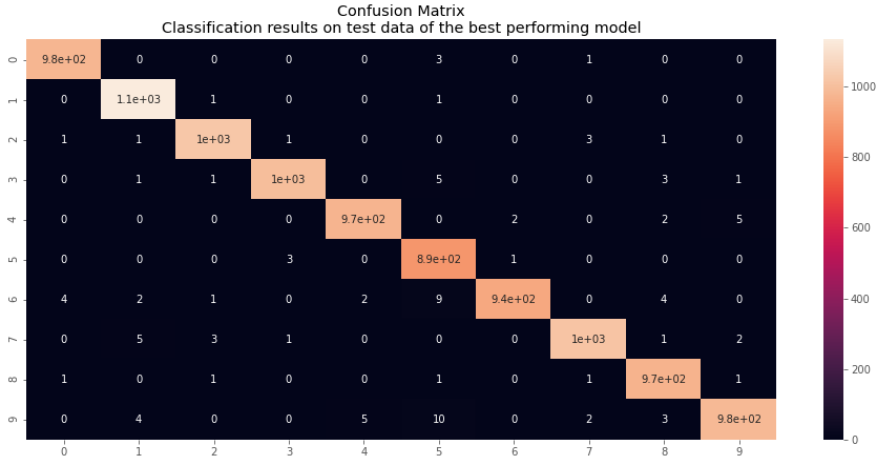
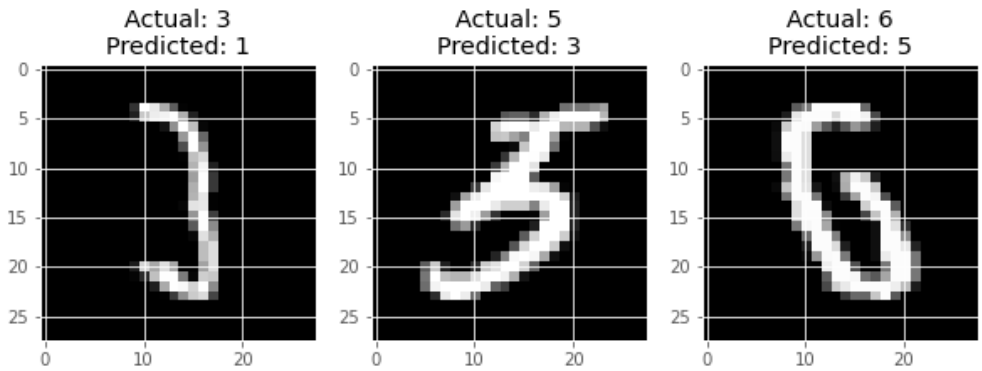


Figure 20: Confusion matrix of the classification results on the test data partition of the best performing model (3 convolution-activation-pooling blocks, 32 feature maps in the first convolution layer, ReLU as activation function, no dropout, with batch normalization, trained with Adam optimizer and a learning rate of 0.001.)

Below we plot some examples of misclassified digits. Looking at the images, it is actually quite understandable to hesitate between 1 and 3 for the first digit, and between 3 and 5 for the second, even for us humans.



2 Semantic segmentation of Biomedical images

Semantic Segmentation is an image analysis procedure in which we classify each pixel in the image into a class. In this assignment task we will segment images of Hematoxylin and Eosin (H&E) stained microscopy slides of colorectal cancer. The dataset (`WARWICK.zip`) is provided including ground truth annotations by expert pathologists. To provide improved performance, the stains have been unmixed resulting in two-channel images (the blue component in the images is constant equal to zero). Our task is to, given an input image, compute an output label-image which has the value one (1) where there is a gland in the image, and zero (0) otherwise.

Exercise 2.1: FCN

> *Design a network for semantic segmentation of the provided data set, e.g., by modifying the network you used for the MNIST classification task as specified below.*

In order to re-purpose a CNN which was developed for a classification task (e.g. the one used for the MNIST task), a few modifications have to be considered:

1. The spatial dimensions of the input which have been reduced by pooling layers have to be recovered by transposed convolution layers to gain exact alignment of class maps and the image.
2. Do not include any fully-connected layers. The very last fully-connected layer may be replaced by a 1×1-convolution.
3. The softmax has to be taken in the channel-direction to give a class prediction per pixel.
4. The loss function has to be adapted to handle a ground truth given by a binary image of the same spatial dimensions as the input image, not a single label for the entire image as in classification.

According to the instructions, we modify the network used in Exercise 1.2 (1). After the last activation layer, we add two transposed convolution layers to get a full-resolution output, with the same spatial dimension as the input image. The classification layer which was a `Linear` layer in our CNN is replaced by a `Conv2d` with a kernel size of 1. The size of the network's output is (batch_size, channels=2, height=128, width=128). The network is predicting 2 channels: the first channel is representing the background i.e class 0 and the second one is representing the gland i.e class 1.

```
class FCN(nn.Module):
  def __init__(self, in_channels=3, out_channels=2):
    super(FCN, self).__init__()

    self.model = nn.Sequential(
      nn.Conv2d(in_channels, out_channels=8, kernel_size=3, stride=1, padding=1),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),
      nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),
      nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
      nn.ReLU(),
      nn.ConvTranspose2d(32, 16, kernel_size=2, stride=2),
      nn.ConvTranspose2d(16, 8, kernel_size=2, stride=2),
      nn.Conv2d(8, out_channels, kernel_size=1), # classification layer
    )
```

```
18
19    def forward(self, x):
20        ...
```

For our segmentation task, we need to compute a pixel-wise cross-entropy loss. In order to do that, we provide the network's output and target in $H \times W$ format to the loss criterion `nn.CrossEntropyLoss()`. More precisely, during training, the loss function will be fed a prediction of shape (batch_size, 2, height, width) and a target of shape (batch_size, height, width). Softmax will be applied inside `nn.CrossEntropyLoss()` and compute a pixel-wise cross-entropy loss.

As for the performance measure, we use Sørensen–Dice coefficient (DSC score):

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

for A and B being the gland segmentation maps of the network and the ground truth annotation, respectively. In our code, it is implemented as the following:

```
1  def compute_DSC_score(prediction, truth, average=True):
2    """Computes the DSC score based on the network's output (prediction) and the ground
     ↪  truth annotation (truth) of the images.
3
4    Args:
5      prediction (tensor): network's linear output (before softmax), of shape
     ↪  torch.Size([batch_size, 2, 128, 128])
6      truth (tensor): the ground truth annotation of the images, of shape
7      torch.Size([batch_size, 128, 128])
8      average (bool): if True, computes the average DSC score of all images, otherwise
     ↪  computes the DSC score for each example
9    """
10
11   prediction = nn.Softmax(dim=1)(prediction)
12   segmentation_map = torch.argmax(prediction, dim=1)
13   numerator = 2 * torch.sum(segmentation_map * truth, dim=(1,2))
14   denominator = torch.sum(segmentation_map, dim=(1,2)) + torch.sum(truth, dim=(1,2))
15   division = torch.div(numerator, denominator)
16   if average:
17     DSC_score = torch.mean(division)
18   else:
19     DSC_score = division
20   return DSC_score
```

The function `compute_DSC_score()` takes as input the linear output of the network, so before applying softmax, and the ground truth annotation. To compute the gland segmentation map from the linear output, we first apply the softmax function channel-wise (which is done with `dim=1`), and then compute the argmax channel-wise too. `segmentation_map` has now the same shape as `truth` and is composed of 0s and 1s, where 0 represents the background and 1 a gland. The set intersection is implemented as an element-wise multiplication. This is valid because the classes' values are 0 and 1. Then the cardinality of a set is simply obtained by summing the elements in the matrices, which correspond to dimensions 1 and 2 (dimension 0 represents the number of samples). An additional argument can be given to the function to decide if we want a single DSC score, the averaged from all examples, or the DSC score per example.

Using Adam optimizer with the default learning rate 0.001 and a mini-batch size of 32, we train our network for 100 epochs. Below we plot the training and test loss and DSC score over 100 epochs. The training loss decreases slowly during the first 20 epochs, then decreases a lot during the next few epochs,

suggesting that the model is learning better. The loss on test data, considered here as the validation data, is also following the same trend but with more fluctuations, probably because there are less samples in the test set. As for the DSC score plot, the two curves look very similar. After about 10 epochs, the training DSC score is higher than the test DSC score, suggesting that the network started to overfit a little. We were still able to reach a DSC score of **0.7**.

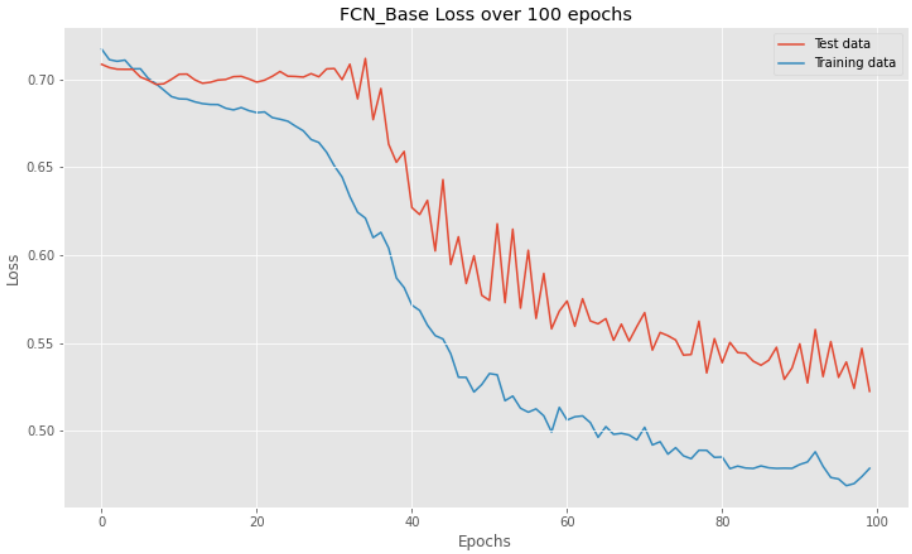


Figure 21: FCN Loss on training and test data vs. number of epochs. The curves follow a similar trend but overall the training curve is lower than the test curve, meaning that the network started to overfit very early in the training process.

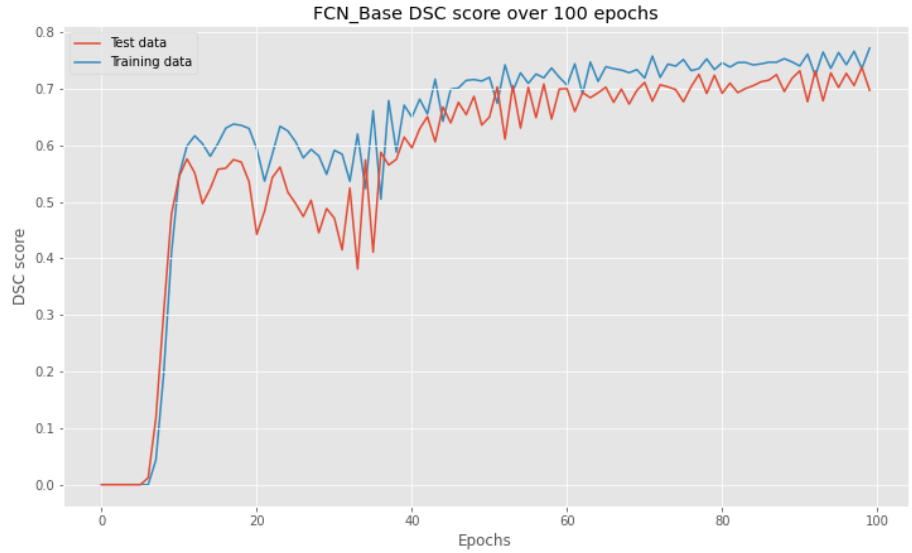
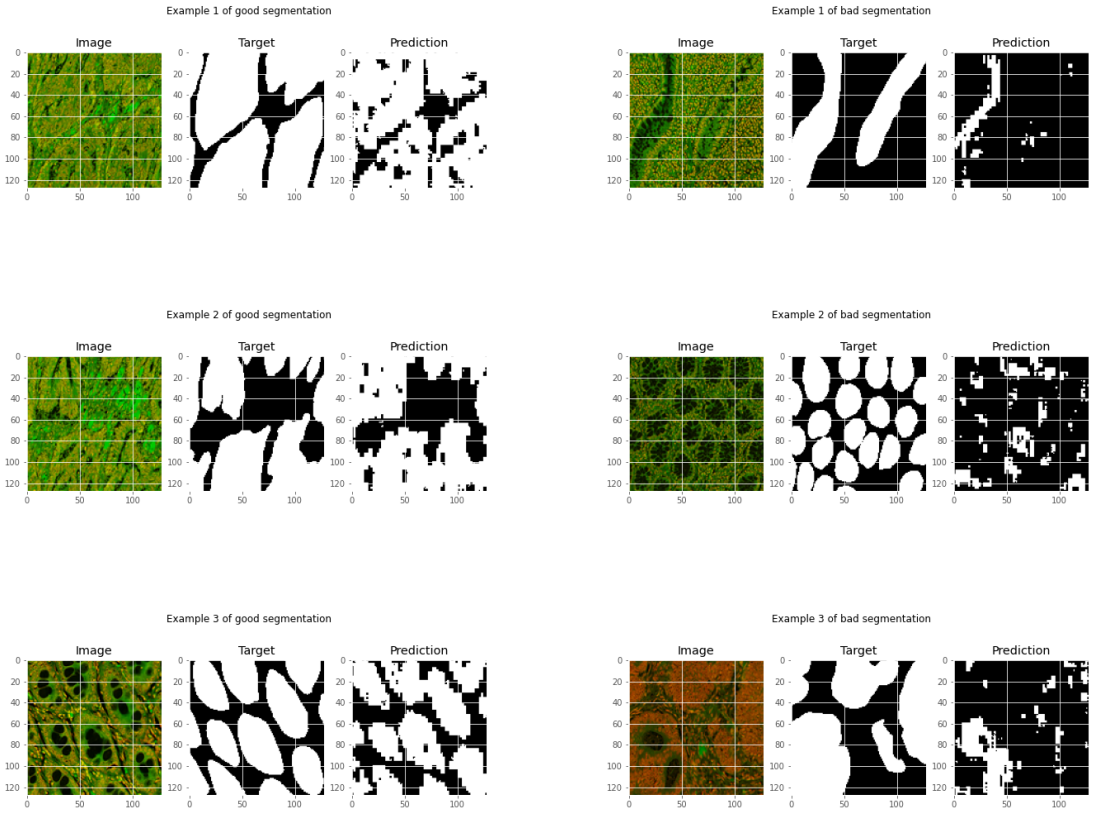


Figure 22: FCN DSC score on training and test data vs. number of epochs. The similitude between the two curves is more pronounced when measuring DSC score. After about 10 epochs, the training DSC score is higher than the test DSC score, suggesting that the network started to overfit a little. We were still able to reach a DSC score above 0.7

Now, we want to visually inspect how the network performs the task on the test data. For that, we evaluate the model on the test data, computed the DSC score for each example in the set (by setting the parameter `average` of our function `compute_DSC_score()` to `False`), and plot the three best and worse predictions. Below are the original image, the target and the network's segmentation map where the network performed well and not very well in terms of the evaluation metric.



The network seems to not perform well for the instances where the glands are darker than the background or when there is not a clear outline separating the gland from the background. When we look at the images with a resulting good segmentation, we can easily see where the glands are whereas in the other instances, it is quite blurry.

Exercise 2.2: Improving the performance on unseen data

Changing the network's architecture.

To improve the performance on unseen data, I first tried to change the network's architecture since the model in itself plays an important role in having good results. Inspired by Ronneberger et al. U-Net [1], I added a convolution layer with kernel size of 3 and ReLU layer after each transposed convolution layer. As batch normalization is often helpful, I also added a batch normalization layer after each convolution layer and before the activation function.

```
class FCN2(nn.Module):
  def __init__(self, in_channels=3, out_channels=2):
    super(FCN2, self).__init__()

    self.model = nn.Sequential(
      nn.Conv2d(in_channels, out_channels=8, kernel_size=3, stride=1, padding=1),
```

```
        nn.BatchNorm2d(8),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),

        nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(16),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),

        nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(32),
        nn.ReLU(),

        nn.ConvTranspose2d(32, 16, kernel_size=2, stride=2),
        nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(16),
        nn.ReLU(),

        nn.ConvTranspose2d(16, 8, kernel_size=2, stride=2),
        nn.Conv2d(8, 8, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(8),
        nn.ReLU(),

        nn.Conv2d(8, out_channels, kernel_size=1), # classification layer
    )

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

The network was trained for 100 epochs, using Adam optimizer with a learning rate of 0.001, and mini-batch size of 32. We were able to achieve a DSC score of **0.8** on the test data, so 0.1 point better than the previous model. However, this result should be nuanced as we can observe in the plots below that the network overfits very quickly, before 20 epochs. The training DSC score is always approximately 0.05 point higher than the test DSC score. This can be explained by the fact that we have little training examples compared to the complexity of network, which is deeper than the previous one.

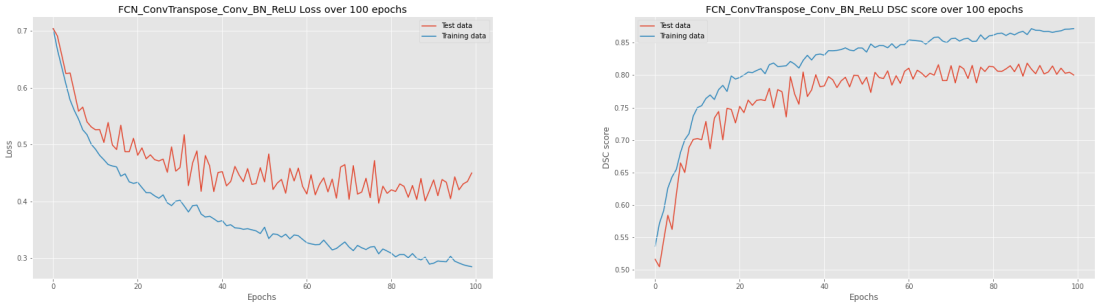


Figure 23: Loss and DSC score on training and test data vs. number of epochs for the FCN with convolution and ReLU layers after transposed convolution layers, and batch normalization. After training for 100 epochs, a 0.8 DSC score on test data was achieved, which is better than the score obtained by the previous network (0.7). However, we can observe that this network overfits very quickly.

Data Augmentation.

To solve the overfitting of our new network, we can actually augment the data by applying several transformations to each training examples (images and labels). The two chosen transformations are

flipping the images and labels in the left-right and up-down direction. This can easily be implemented using `numpy` functions `fliplr` and `flipud`.

```
flipped_lr_image = np.fliplr(image)
flipped_ud_image = np.flipud(image)
```

Below are some examples:

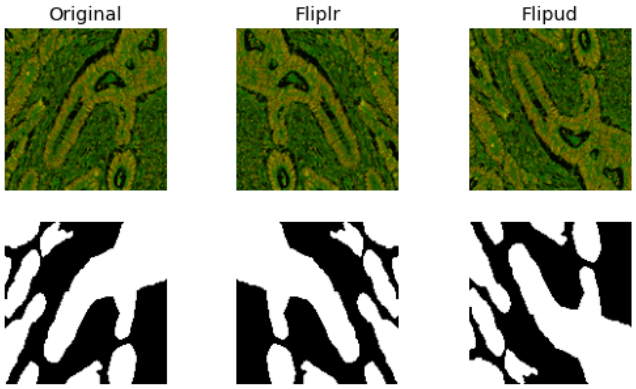


Figure 24: Flipped images and labels in the left-right and up-down direction.

By applying flipping to each image and label in the training set, we triple the size of the training dataset, from 85 to 255. I also tried to apply two others image augmentation techniques, rotation of 45 degrees (`skimage.transform.rotate`) and adding random noise (`skimage.utils.random_noise`), but it strangely resulted in black images for some of the instances. Since it would have distort the training, I only decided to flip the images, which is actually an extension of rotation. We trained the new network, this time on the augmented data, and validated on the test set. We were able to achieve a **0.82** DSC score on test data, but the network still overfits.

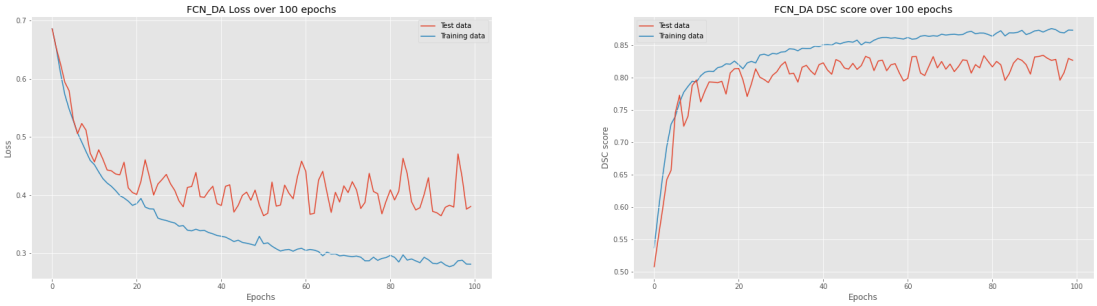


Figure 25: Loss and DSC score on augmented training and test data vs. number of epochs for the FCN with convolution and ReLU layers after transposed convolution layers, and batch normalization. After training for 100 epochs, a 0.82 DSC score on test data was achieved, but the network still overfits.

Hyperparameters tuning with the augmented data and early stopping.

Previously, we evaluated the network on the test set, which was also used to validate the network during training. With this method, we do not know how the algorithm actually performs on new data. Hence, the next technique I tried is to split the augmented training data into a new train set and a validation set, so that we can use the validation set to perform hyperparameters tuning and the test set to evaluate the network on real unseen data and get more accurate results.

We take 60 examples of the augmented training dataset, after shuffling it, to make the validation set the same size as the test set, and the rest of the samples will constitute the new training set (195). We can now properly evaluate our model after training on unseen data using the following function. It first compute predictions on the provided test data and then compute the DSC score against the test target.

```
def evaluate(model, X_test, Y_test):
  predictions = model(X_test)
  score = compute_DSC_score(predictions, Y_test)
  return score
```

We also implemented early stopping in our `train_model()` function to prevent the model from overfitting too much. For that, we introduce a parameter `epsilon`, which indicates the tolerance between the validation DSC score of the previous and current epoch. So for example, if we set `epsilon = 0.03`, then we stop the training if the absolute difference between the validation DSC score of the previous epoch and the current one is greater than 0.03. We also do not want to stop the training too early in the process, so we add another condition to stop it from 20 epochs.

```
def train_model(...):
    ...
    # Early stopping
    if len(test_scores) >= 20 and abs(test_scores[-1] - test_scores[-2]) > epsilon:
        break
    ...
```

The first hyperparameter we wanted to tune is the batch size. We trained the network for different batch sizes in `[8, 16, 32, 64, 128]` for 100 epochs, using Adam optimizer and a learning rate of 0.001. With `epsilon = 0.03`, no early stopping happened for all models, meaning that the gap between the training and validation DSC score curves were not significant. Below are the DSC score we were able to obtain on the test data.

Batch size	DSC score on test data
8	**0.8359**
16	0.8150
32	0.8036
64	**0.8203**
128	0.7979

Table 1: DSC score on test data depending on the batch size

The network trained with mini-batch size 8 and 64 performed the best on unseen data. However, if we compare the training and validation curves, it seems that the network overfits when we set a batch size of 8, so in the following, we choose a batch size of 64.

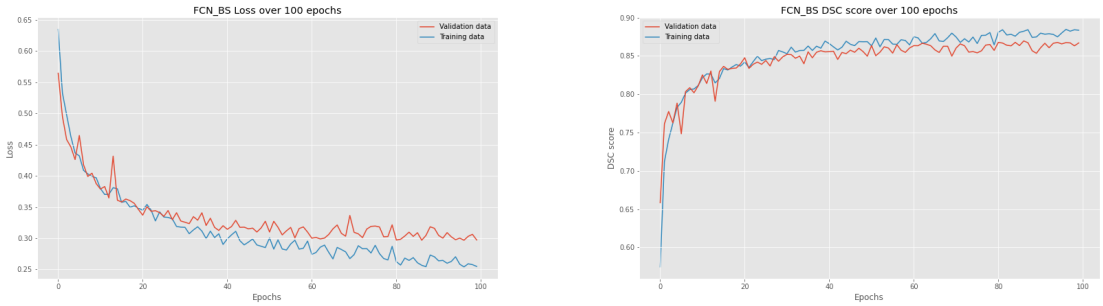


Figure 26: Training and validation loss vs. number of epochs, with mini-batch size of 8. The network seems to overfit a little after about 30 epochs.

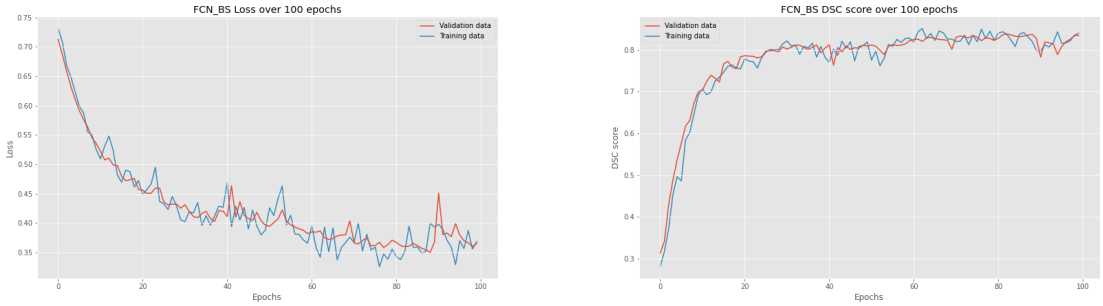


Figure 27: Training and validation loss vs. number of epochs, with mini-batch size of 64. Compared to the above results, the network is not overfitting or underfitting.

We also try to tune the learning rate of Adam optimizer. For that, we trained the same network for different values of learning rates in `[1e-2, 1e-3, 1e-4, 1e-5]`.

Learning rate	DSC score on test data
0.01	0.8078
0.001	**0.8230**
0.0001	0.7132
0.00001	0.6411

Table 2: DSC score on test data depending on the learning rate of Adam optimizer

The results confirm that the default learning rate is a good choice as it gives the best results on unseen data.

Best model.

Our best model is the second network, with convolution and ReLU layers after each transposed convolution layer, and batch normalization after each convolution layer. Based on our experiments, it gives the best results (best DSC score and not too overfitting) on the test data when using a mini-batch size of 64 and Adam optimizer with a learning rate of 0.001. We trained the network for more epochs, and more precisely for 200 epochs. We also set `epsilon` to 0.05 but it did not stop the learning as the gap between the training and validation DSC scores was small across all epochs. We reached a final DSC score of **0.83** on the test data.

To see how our best model performed on the images where the previous first network did not give satisfying results, we plot the segmentation maps of the same images below.

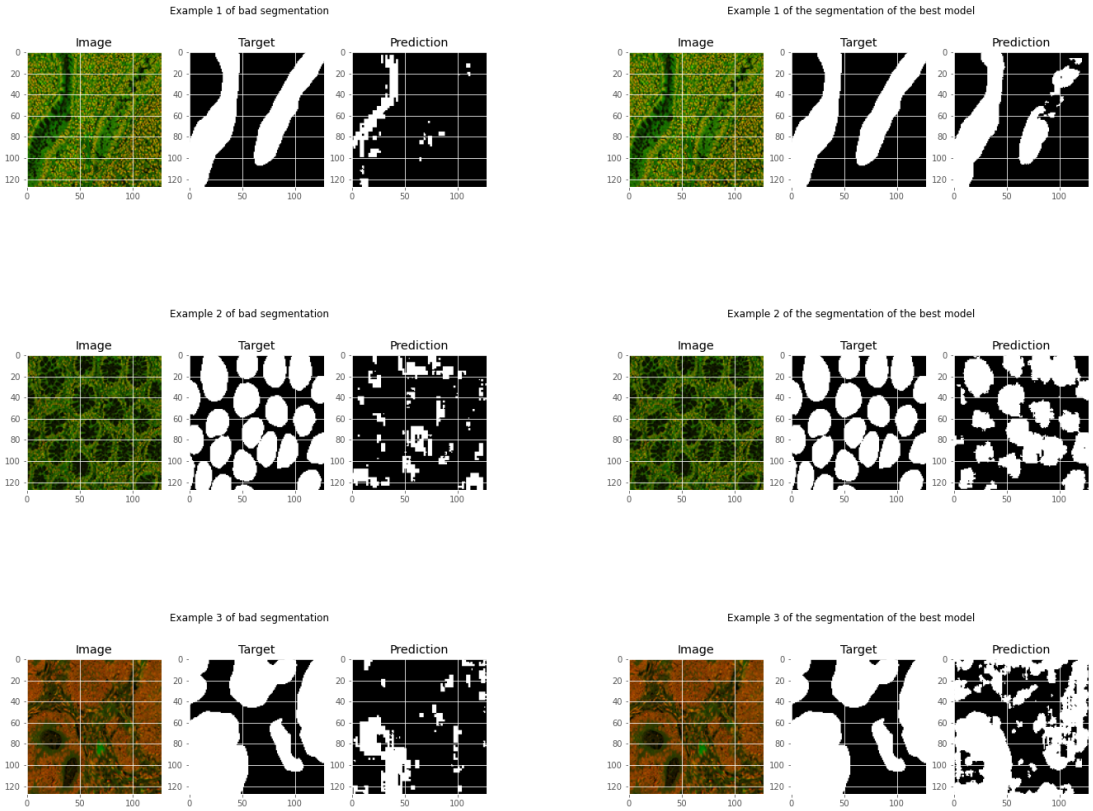


Figure 28: On the left, segmentation maps of the first network of Exercise 2.1 (2) where it did not perform well. It was trained for 100 epochs with mini-batch size of 32, and Adam optimizer with a learning rate of 0.001. It reached a DSC score of **0.7** on test data which was also used as validation set during the training. On the right, segmentation maps of the same images, but of the best trained model, which reached a DSC score of **0.83** on test data which was not used as validation set during the training. We can definitely see an improvement in segmentation.

For some instances, the network still finds it difficult to differentiate glands from the background.

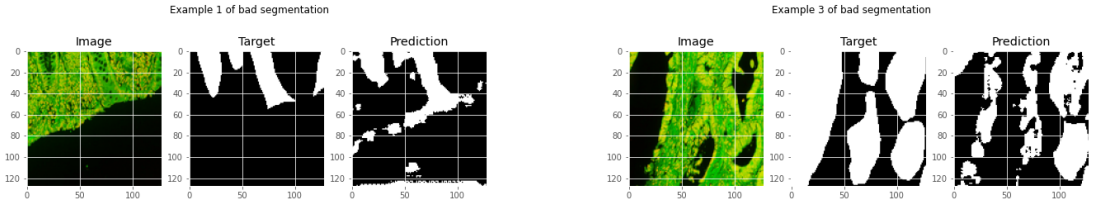


Figure 29: On the left, the glands are not black but on the right, they are, which can cause confusion and makes it challenging to segment.

References

[1] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.