Homework6

Q2.
Point b.
I chose as hyperparameters the following ones:

- Learning rate: 0.001
- Number of epochs: 20
- Batch size: 64

The number of epochs is a tradeoff between the model performance and the computational complexity. I also tried different learning rates, but I chose 0.001 since higher ones (like 0.005, 0.01) seemed introducing too many oscillations in the loss curve over all epochs.
The results obtained with this configuration are shown in Figure 1 with a final accuracy of 89.1%.

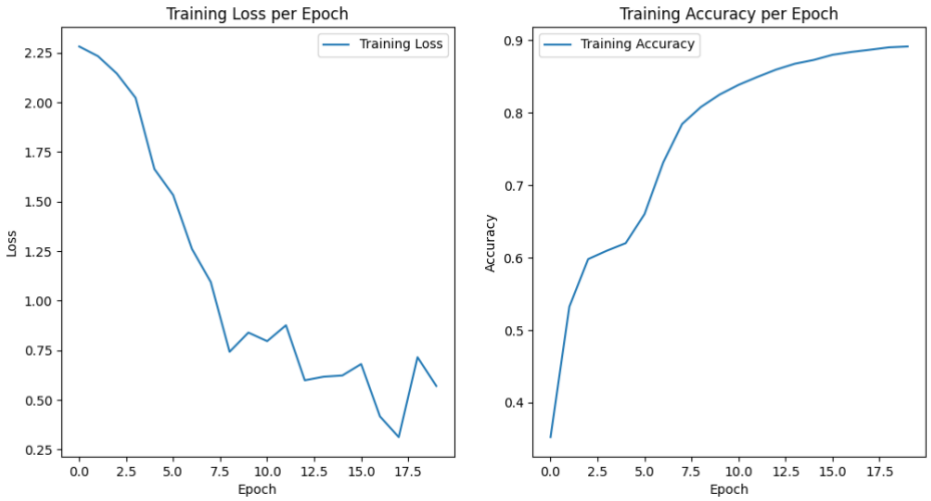


*Figure 1*

Q3.

Point a.
The code to define the network with the described architecture is shown in Figure 2.

```python
# Network structure
input_channels_first_layer = 1
output_channels_first_layer = 20
kernel_size_first_layer = 4
stride_first_layer = 1
input_channels_second_layer = 20
output_channels_second_layer = 20
kernel_size_second_layer = 4
stride_second_layer = 2
kernel_size_max_pooling = 2
stride_max_pooling = 2
# Output after the first convolutional layer: 20x25x25
# Output after the second convolutional layer: 20x11x11
# Output after the max pooling layer: 20x5x5
input_size = 20*5*5 #500
hidden_size = 250
output_size = 10

# Define networks
class ConvolutionalNeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.model_CNN = nn.Sequential(
            nn.Conv2d(in_channels=input_channels_first_layer, out_channels=output_channels_first_layer, kernel_size=kernel_size_first_layer, stride=stride_first_layer),
            nn.ReLU(),
            nn.Conv2d(in_channels=input_channels_second_layer, out_channels=output_channels_second_layer, kernel_size=kernel_size_second_layer, stride=stride_second_layer),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=kernel_size_max_pooling, stride=stride_max_pooling)
        )
        self.model_fully_connected_NN = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, output_size)
        )

    def forward(self, x):
        x = self.model_CNN(x)
        x = x.view(x.size(0), -1)
        logits = self.model_fully_connected_NN(x)
        return logits
```

*Figure 2*

Point b.
After training such network we get the results shown in Figure 3. We can see that we can achieve way better results compared to Q2, using the same hyperparameter values, as the final accuracy is 96.9%.

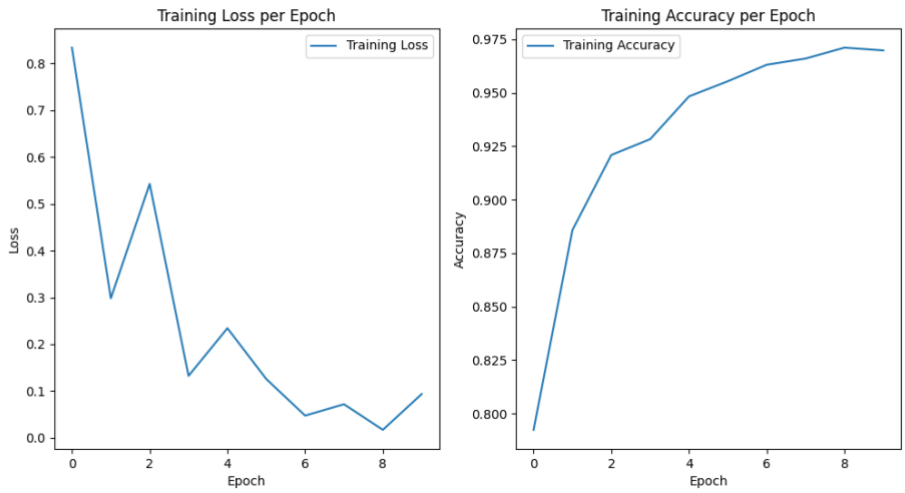


*Figure 3*

Point c.
The new network is defined as in Figure 4.

```python
class ConvolutionalNeuralNetworkAdvanced(nn.Module):
    def __init__(self):
        super().__init__()
        self.model_CNN = nn.Sequential(
            nn.Conv2d(in_channels=input_channels_first_layer, out_channels=output_channels_first_layer, kernel_size=kernel_size_first_layer, stride=stride_first_layer),
            nn.ReLU(),
            nn.Conv2d(in_channels=input_channels_second_layer, out_channels=output_channels_second_layer, kernel_size=kernel_size_second_layer, stride=stride_second_layer),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=kernel_size_max_pooling, stride=stride_max_pooling)
        )
        self.model_fully_connected_NN = nn.Sequential(
            nn.Dropout(dropout_first_layer),
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout_second_layer),
            nn.Linear(hidden_size, output_size)
        )

    def forward(self, x):
        x = self.model_CNN(x)
        x = x.view(x.size(0), -1)
        logits = self.model_fully_connected_NN(x)
        return logits
```

*Figure 4*

The new hyperparameters used are:

- Dropout: 25%
- Weight decay: 1e-4

I tried a dropout of 50%, but then I realized that a smaller dropout gives better results in this case, so I chose 25%. I also tried to add dropout regularization to just one of the two layers, but the configuration that produced the best results is the one that applies dropout regularization to both layers. Applying dropout regularization only to the second layer achieved a better accuracy compared to applying it only to the first one, though.
For the weight decay I also tried different values around 1e-4, like 1e-2 and 1e-6, but they both lead to worse values of accuracy (around 93%).
The obtained results are shown in Figure 5, where the final accuracy is 96.8%.

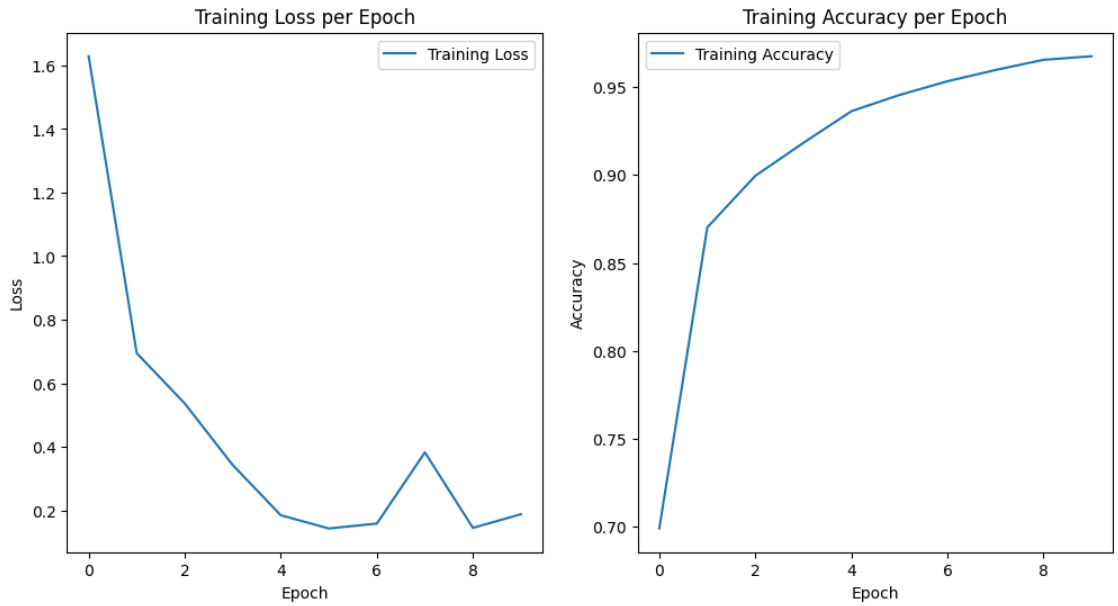


*Figure 5*

Point d.
The new network is defined as in Figure 6.

```
class ConvolutionalNeuralNetworkFinal(nn.Module):
    def __init__(self):
        super().__init__()
        self.model_CNN = nn.Sequential(
            nn.Conv2d(in_channels=input_channels_first_layer, out_channels=output_channels_first_layer, kernel_size=kernel_size_first_layer, stride=stride_first_layer),
            nn.ReLU(),
            nn.BatchNorm2d(output_channels_first_layer),
            nn.Conv2d(in_channels=input_channels_second_layer, out_channels=output_channels_second_layer, kernel_size=kernel_size_second_layer, stride=stride_second_layer),
            nn.ReLU(),
            nn.BatchNorm2d(output_channels_second_layer),
            nn.MaxPool2d(kernel_size=kernel_size_max_pooling, stride=stride_max_pooling)
        )
        self.model_fully_connected_NN = nn.Sequential(
            nn.Dropout(dropout_first_layer),
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout_second_layer),
            nn.Linear(hidden_size, output_size)
        )

    def forward(self, x):
        x = self.model_CNN(x)
        x = x.view(x.size(0), -1)
        logits = self.model_fully_connected_NN(x)
        return logits
```

*Figure 6*

The hyperparameters used are:

- Learning rate: 0.001
- Number of epochs: 20
- Batch size: 64

I tried to reduce the learning rate to 0.01, but the oscillations in the average loss per epoch increased a lot, without bringing a significant improvement in the final result. On the other hand, reducing the learning rate would smooth the convergence of the model, but it would also require running for more epochs to get better results. For these reasons, I kept the previous value of 0.001.
Concerning the number of epochs, it's easy to see from the plot that the accuracy stabilizes around the last epochs, so I decided to keep 20 epochs since more epochs won't get significantly better results and also considering that the time taken to run the code increases with the number of epochs.
Finally, I tried to increase the batch size to 128 and decrease it to 32 to see the effect on the final result. I observed that a large batch size (128) makes the code run faster (less batches to analyze), but also gives worse results than the previous 64 batch size. With a batch size of 32, I observed a similar behavior: despite taking longer due to the smaller batches, it didn't bring a significant improvement with respect to 64. In the end, I decided to go for a tradeoff between the two opposite possible choices, choosing 64.

The obtained results are shown in Figure 7, where the final accuracy is 98.8%.

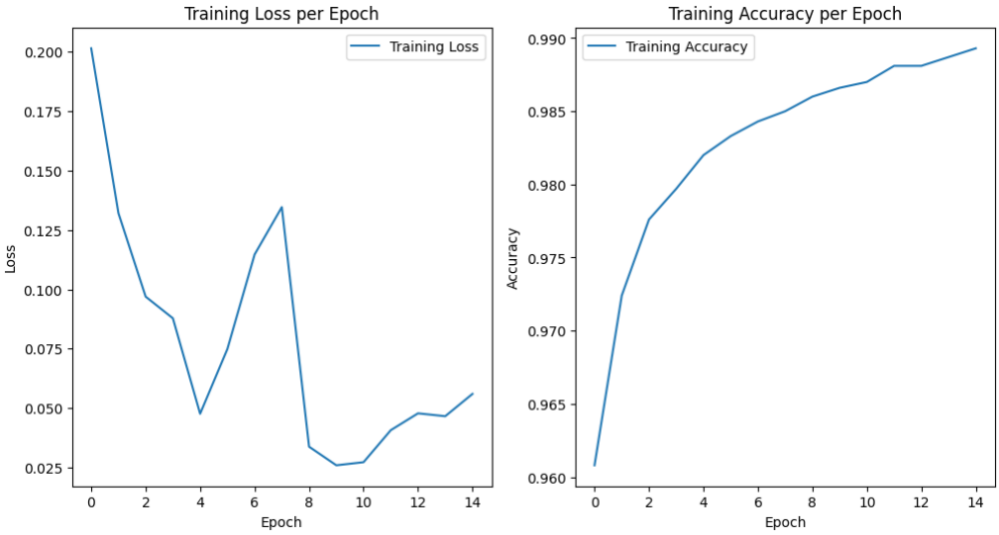


*Figure 7*

To further improve the results, we can consider other important aspects:

- Weight initialization: we know from the theory that one of the best initialization methods is to use random values drawn from a distribution with small variance. Pytorch already implements this type of initialization, though.
- Early stopping: to avoid overfitting we can monitor the loss and when it increases significantly for consecutive epochs it's better to stop the computation earlier. This should improve the overall results.

By implementing early stopping I obtained the results shown in Figure 8, achieving an accuracy of 98.2%.

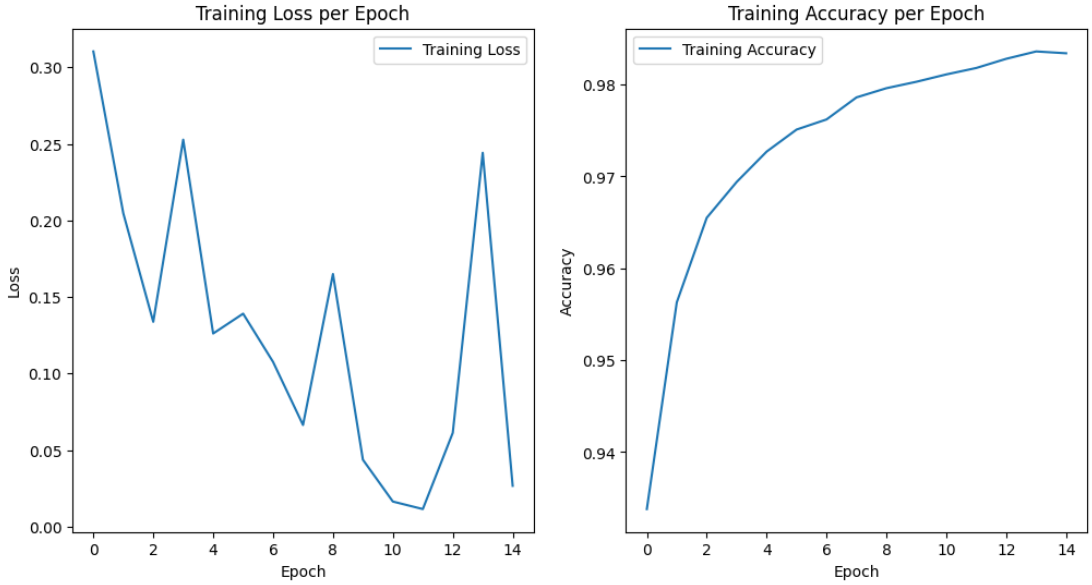


*Figure 8*

From the plots it's clear that the early stopping method is correctly working since the loss increases in the last three epochs, so it stops. The results are not as good as before, though. This could be due to many reasons: one of them could be the way I decided to implement the early stopping method, which could, for sure, be improved by adding more complex reasoning behind it. Also, in my implementation there is a parameter that represents the number of consecutive epochs we need to observe an increased loss to stop the computation, which is an additional hyperparameter which needs to be tuned at best.

Since early stopping prevents overfitting by stopping the computation when the loss increases for a fixed number of consecutive epochs, I also tried to increase the learning rate to 0.01 to see if the model is able to quickly get to a very good result and if the early stopping method can stop the computation before it starts oscillating too much.
In Figure 9 are shown the obtained results with this last change, reaching a final accuracy of 98.9% that equals the results obtained without early stopping with a learning rate of 0.001. Of course, this solution with a learning rate of 0.01, is less stable and reaches different results from run to run since the updates from one epoch to the other are higher.

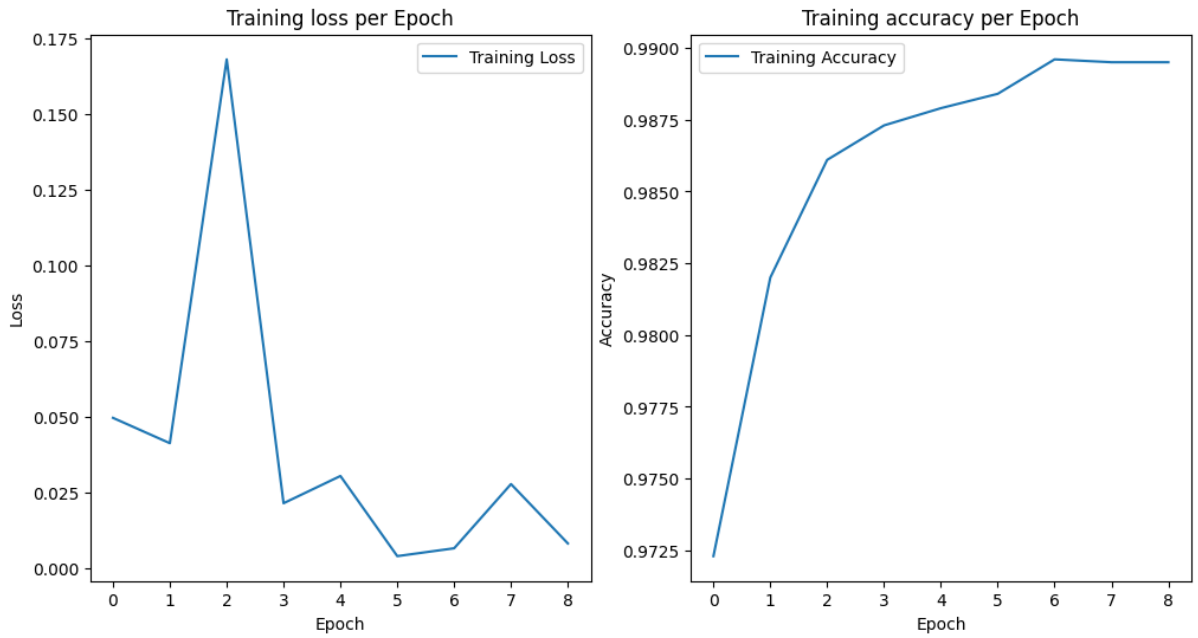


*Figure 9*