Deep Learning – 1RT720

Report for Hand-in assignment 1

Caleb Caulk

March 3, 2025

1 Introduction

This document is the report for Hand In assignment 2 for Deep Learning. In this assignment we were tasked with constructing several neural networks to evaluate the MNIST dataset on hand written digits. First, we were tasked with creating a fully connected neural network and compare our results with the one we created in assignment 1. Then we were tasked with creating a Convolutional Neural Network with a provided architecture schema, and we were tasked with modifying that to see how the performance changes.

The full code is available in Appendix A.

2 Template for reporting Hand-in assignment 1

Exercise 1. *Implement a neural network in Pytorch*

In exercise 1 we implemented a neural network using the Pytorch library. In this exercise we were asked to make a mult-layered fully connected neural network that is identical to one of the networks that we implemented in assignment 1. I implemented a deep neural network with the following architecture that is identical to the architecture I used in assignment 1. The deep neural network had:

- 2 hidden layers
- 50 nodes per hidden layer
- Input dimension 784
- Output dimension 10
- ReLU for all activation functions
- Mini-batch of size 128
- Training time was 150 epochs
- Learning rate of 0.01

Exercise 1.a *Compare the reached performance with your Assignment 1 results; do you observe similar accuracy?*

Both models reach over a 98% training accuracy and over a 97% test accuracy, so both models are very similar in accuracy. Interestingly enough the Pytorch implementation had a significantly lower loss function value vs. my Numpy implementation which may suggest that the PyTorch implementation is more certain in it's predictions.

Exercise 1.b *How many times faster/slower was your own implementation? (Make sure to indicate whether you are running on a CPU, Google Colab, or with enabled GPU support if a suitable graphics card is available).*

I ran both of training loops on my local computer's CPU and the Pytorch implementation was around 1.4 times slower than my Numpy implementation. The times can be seen in Figure 1 for the numpy implementation and Figure 2 for the Pytorch implementation. I'm not sure why this is much slower, but maybe it is due to how I store the loss and accuracy values when training to make the training and test curves.

Exercise 1.c *Provide a learning curve plot.*

Note that the loss and accuracy for the training data set for each epoch was calculated by averaging all the batch losses and averages, and this resulted in smoother plot lines compared to those in assignment 1. The test loss and accuracy were tested at the end of each epoch on the entire test dataset.

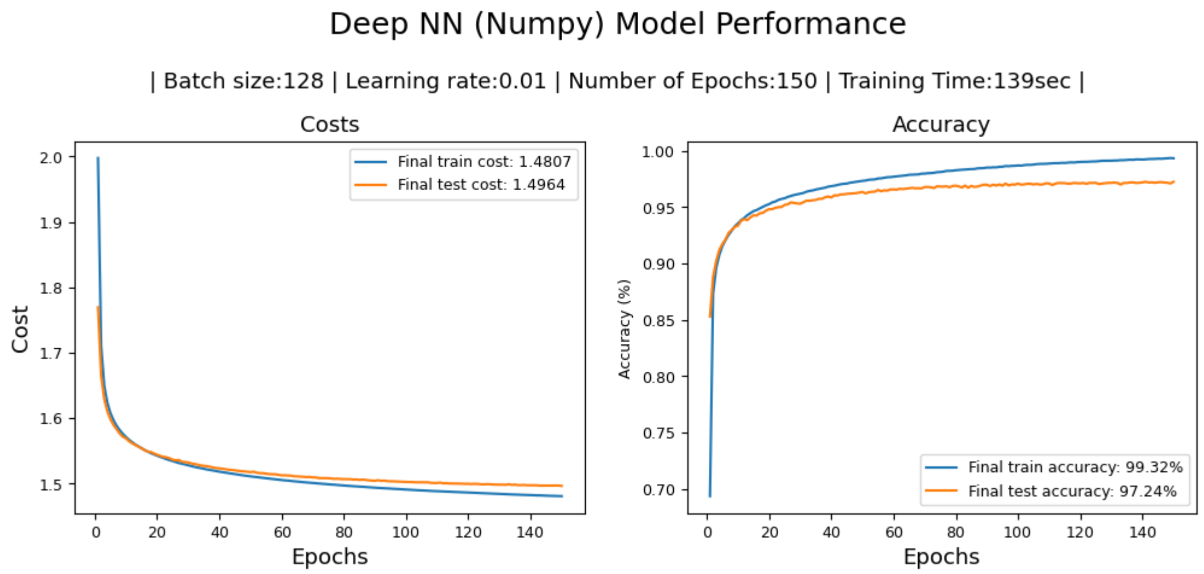


Figure 1: Cost and Accuracy for my Numpy implementation of a deep Neural Network

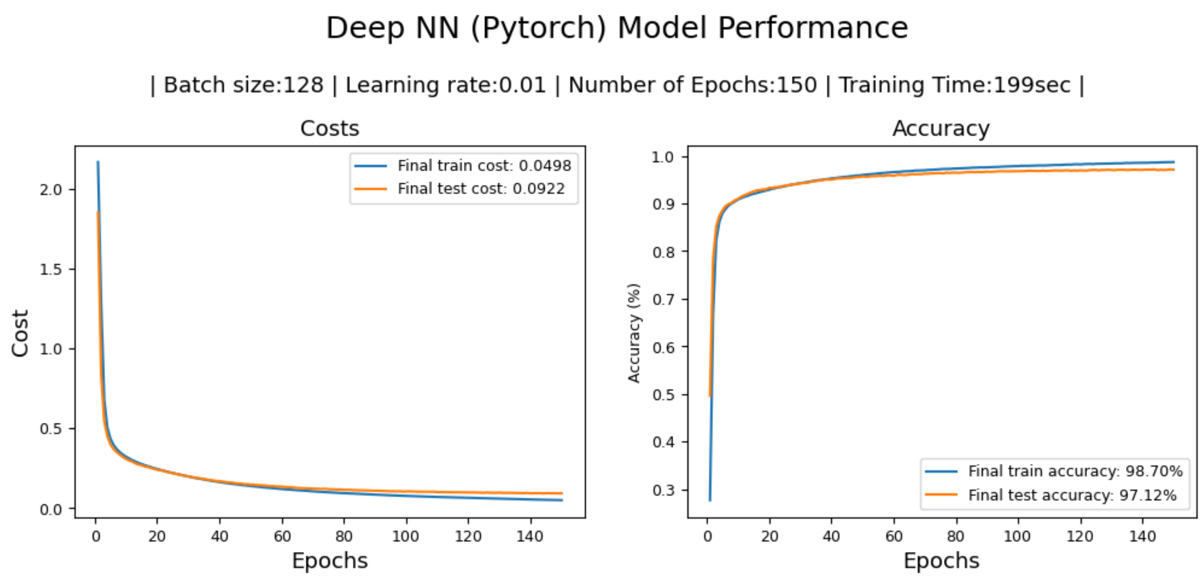


Figure 2: Cost and Accuracy for my PyTorch implementation of a deep Neural Network

Exercise 2 *Code a Convolution Neural Network*

In this exercise we are tasked with implementing a CNN network with architecture described in the assignment document. Also note from this point forward I ran all Neural Networks on my local laptop's GPU to speed up computation time.

Exercise 2a *How many learnable weights does this network contain? Compare with how many weights you had in the previous exercise.*

This model has 21578 learnable parameters which is about half the number of weights as the model in the previous exercise. The fully connected neural network with two hidden layers has 42310 learnable parameters.

Exercise 2b *Provide a learning curve plot.*

The learning curve plot for the CNN network is in Figure 3. There we see that the test accuracy is above 98%

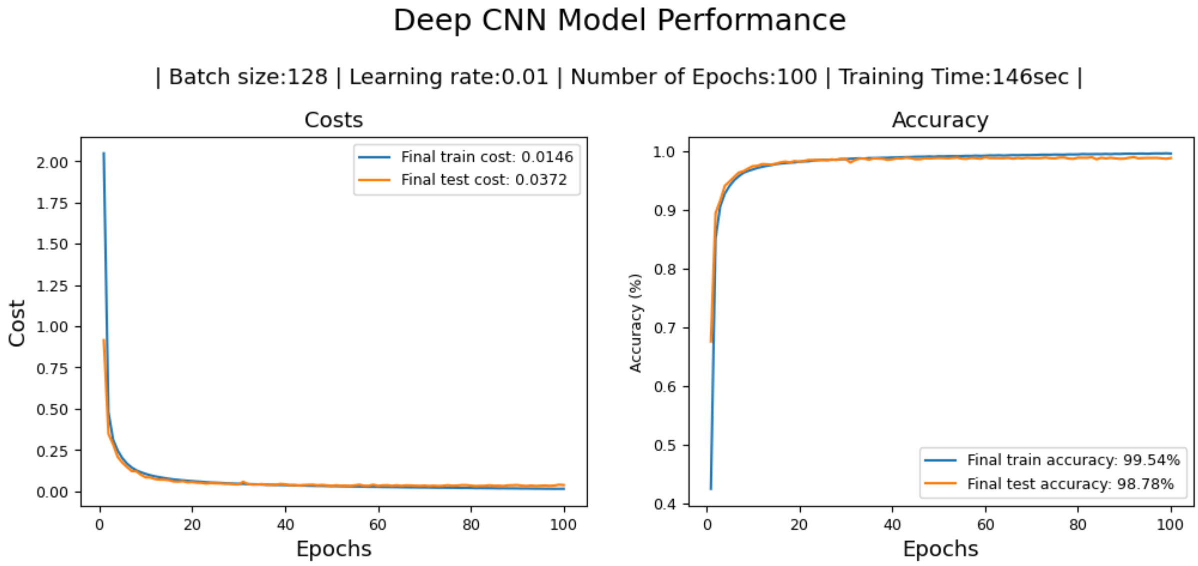


Figure 3: Cost and Accuracy for my implementation of a deep Convolutional Neural Network

Exercise 3 *Swap the order of max pooling and activation function. What do you think will happen?*

In this exercise we are tasked swapping the order of the ReLU activation function and the pooling layers of the CNN. My hypothesis is that this will not affect the models performance because ReLU clips all values less than zero to zero then the max pooling would take the max value. The only case where something interesting happens is when the max value was negative, ReLU would make it zero then max pooling chooses zero. If the order is switched we take the max negative number but ReLU still makes it zero so nothing changes. If the max number is non negative or zero it stays as is and if it is negative it gets clipped to 0 so the order of ReLU and max pooling doesn't matter.

Exercise 3a *Does the change in order affect the model's performance, or does it have no impact? With the changed order, how long does the training take, and what is the final accuracy? Please also provide a learning curve plot.*

The order here did not affect the model's performance. The training takes a little longer if there is max pooling before Relu. The training took around 200 seconds for 100 epochs; whereas, before training took only 146 seconds for the same number of epochs. The final test accuracy was 98.74% and the learning curve can be seen in Figure 4

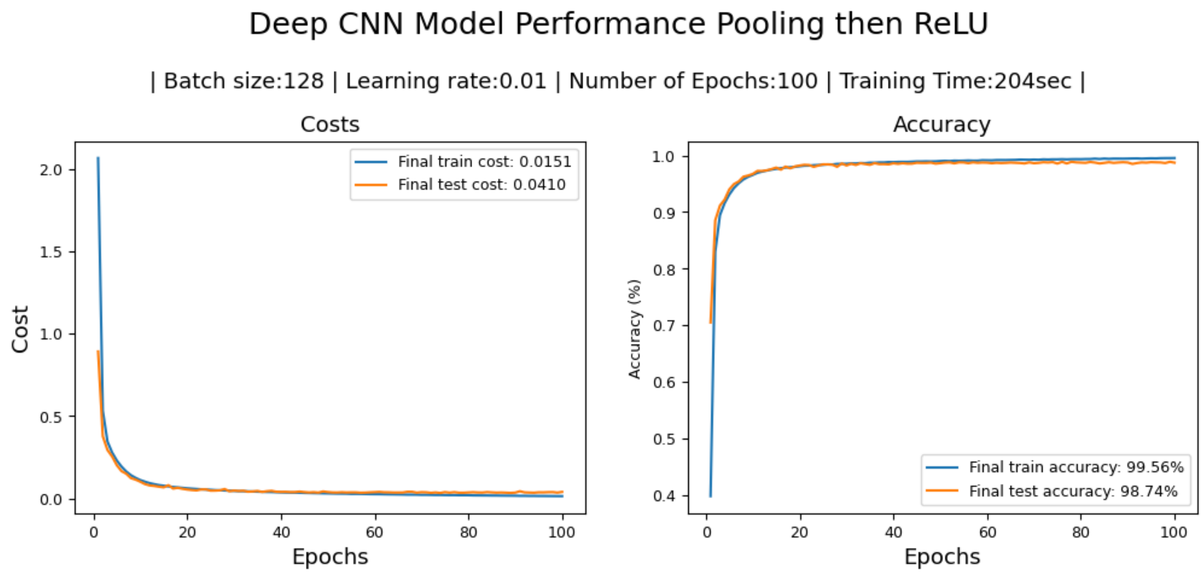


Figure 4: Cost and Accuracy for pooling being executed before ReLU

Exercise 3b *Instead of using ReLU as the transfer function, use the hyperbolic tangent activation function (keeping order still swapped). How long does the training take and what is the final accuracy? Provide a learning curve plot. The difference is probably more distinct here, why?*

Switching out the activation function from ReLU to Tanh makes training a little faster. As shown in figure 5 the accuracy curve is stretched more and one can see it learns a bit slower. With the Tanh activation function it took around 50 epochs to get 98% accuracy versus with ReLU it took around 20 epochs. Training time with Tanh was 162 seconds and the final test accuracy was still really good at 98.7%. This stretching is probably due to Tanh clipping values between [-1,1] which can slow learning.

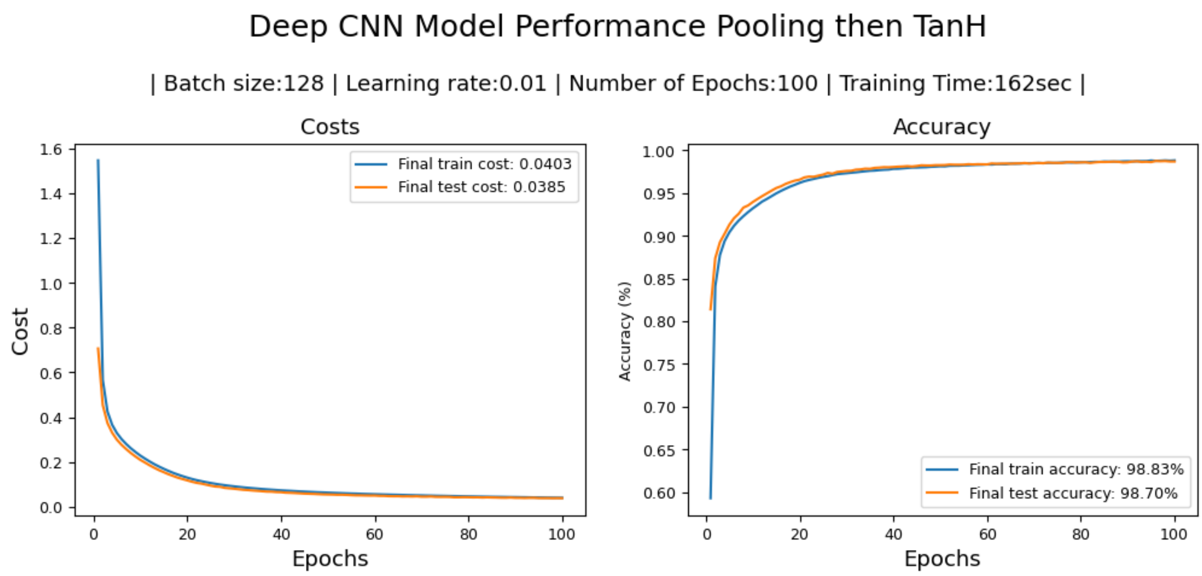


Figure 5: Cost and Accuracy for pooling being executed before TanH

Exercise 3c *What are your conclusions with regard to this?*

My conclusion is that for this specific task ReLU provides a quicker time to learn than Tanh. This is probably due to Tanh restricting values between [-1,1], so when max pool is used the largest value that can be evaluated is 1. ReLU on the other hand maps values between [0, ∞], so when max pool is used the largest value can be much greater than 1 which allows the model to be more expressive and therefore have a faster time to learn good parameters.

Exercise 4 *Use the ADAM optimizer.*
Now change the optimizer and train 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? Provide a learning curve plot.

In Figure 6 you can see the cost and accuracy curves of training the CNN from Exercise 2 with the Adam optimizer instead of the SGD optimizer. Here I had to decrease the learning rate because a learning rate of 0.01 caused to much jumping around. As you can see Adam managed to get better results very quickly. Within the first 10 epochs the model was basically fully trained since the test accuracy was around 99% (the final test accuracy is 99.09%) and the cost was near a minimum of around 0.04. As the model was trained longer, the cost plot shows over training as the cost for the test set slowly increases, but interestingly enough the accuracy remained relatively stable, but still the cost plot indicates that 100 epochs was too long and the model began to over fit.

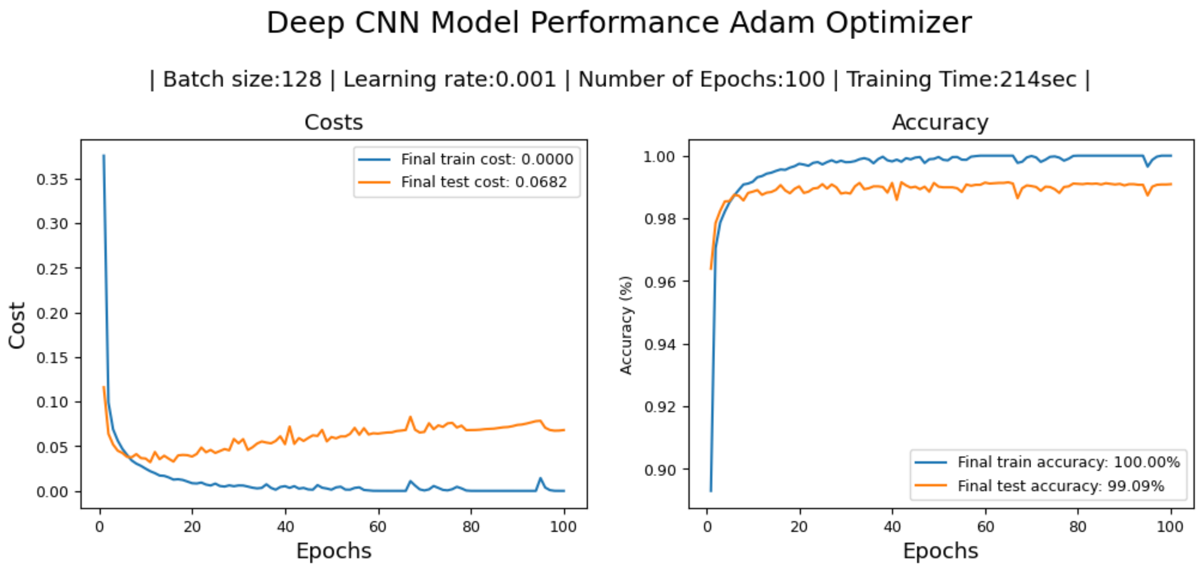


Figure 6: Cost and Accuracy using the Adam optimizer

Exercise 5 *Residual connections.*
Implement residual connections like described in Section 11.2.1 in the UDL course book in your architecture. Evaluate if this improves the performance. In particular, check if the inclusion of residual connections allows training of deeper networks by replacing each convolution+activation pair in your architecture with a block of two or three similar pairs, where the residual connection bridges over each such block. Check the training speed with and without the residual connections in place (for otherwise identical architectures). Provide a learning curve plot.

Adding 9 extra residual layers (3 for each convolution filter type) with SGD as the optimizer did improve the performance a little bit. As shown in Figure 7 the test accuracy is almost 99% which is an improvement compared to all other modifications except for using Adam as the optimizer. The training accuracy is also 100%, which was also only achieved with using Adam as the optimizer. The training time with residuals didn't significantly increase which is interesting because the depth of the network went from 3 hidden layers (not including pooling) to 12 hidden layers. When we compare the training curves of residual architecture versus the same architecture without residual connections, we see a very large difference. Figure 8 shows the training curve of an identical network architecture to that of the deep residual network used to produce the results in Figure 7 except it doesn't contain the residual connections. In Figure 8 we see that a network without residual connections and run with SGD optimizer completely fails. The final cost is very high and the final test accuracy is barely better than random guessing. This would indicate that the implementation of residual connections allows training of deeper networks and an improvement of accuracy.

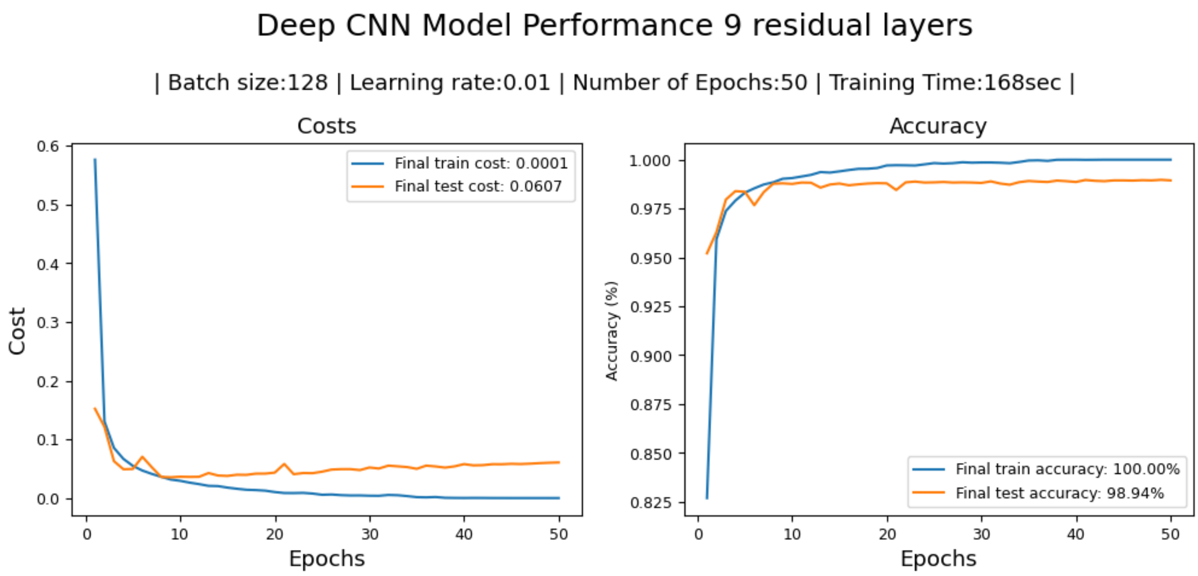


Figure 7: Cost and Accuracy using the 9 extra residual layers and using the SGD optimizer

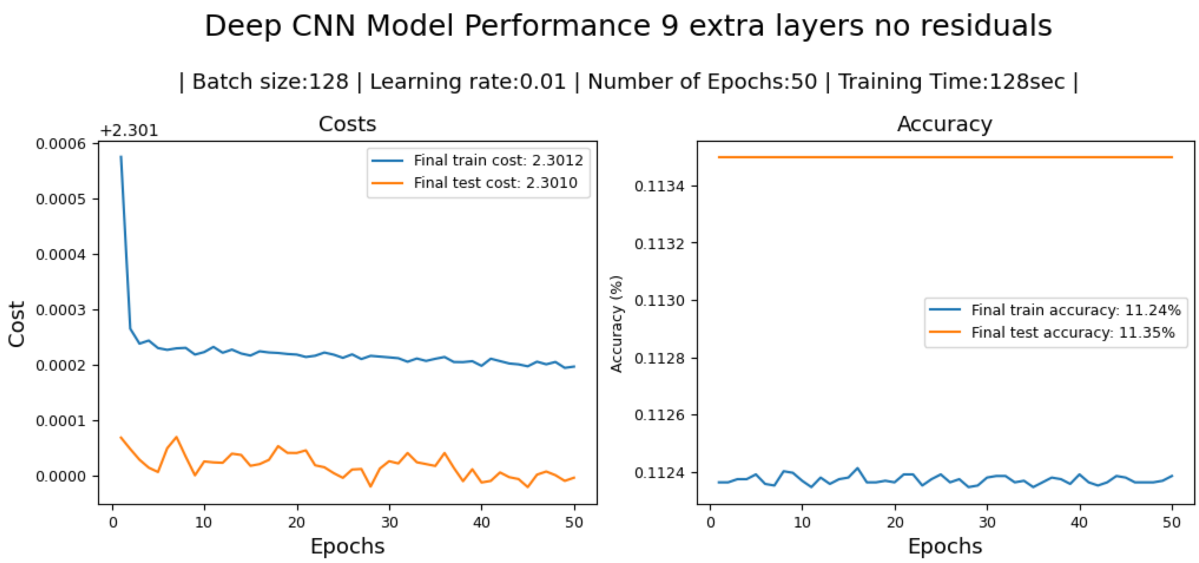


Figure 8: Cost and Accuracy using the 9 extra layers and using the SGD optimizer. No residual connections

Exercise 6 *CNN with three variations.*
Try three 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. At least one of these changes has to be a regularization approach. How good

performance do you manage to reach by tweaking your learning setup?
Exercise 6a *First variation*

For the first variation I will take the residual network architecture from exercise 5 and add batch normalization. I am adding batch normalization as a regularization technique and for stable forward propagation and to use a higher learning rate [Pri23]. I used SGD as the optimizer, cross entropy as the loss function, learning rate of 0.025, 128 mini batch size, and trained for 50 epochs (Training for longer took a lot of time). The network architecture is as follows.

- Input
- 1 Convolutional layer to go from 1 channel to 8
- 3 Residual Convolutional layers from 8 channels to 8 channels with batch normalization
- Max pooling with stride 2
- 1 Convolutional layer to go from 8 channels to 16
- 3 Residual Convolutional layers from 16 channels to 16 channels with batch normalization
- Max pooling with stride 2
- 1 Convolutional layer to go from 16 channels to 32 channels
- 3 Residual Convolutional layers from 32 channels to 32 channels with batch normalization
- Relu to output

The learning curve for this network is displayed below in Figure 9. As you can see adding batch normalization to the residual connections improved performance and got up to a 99.18% test accuracy; however, this model might have started to overfit because in the cost graph we can see the test cost slowly increasing. This indicates the model may be overfitting and there is a 100% training set accuracy and almost 0 training cost, which further indicates the model may be overfitting. Another thing to note is this took significantly longer to run at 528 seconds.

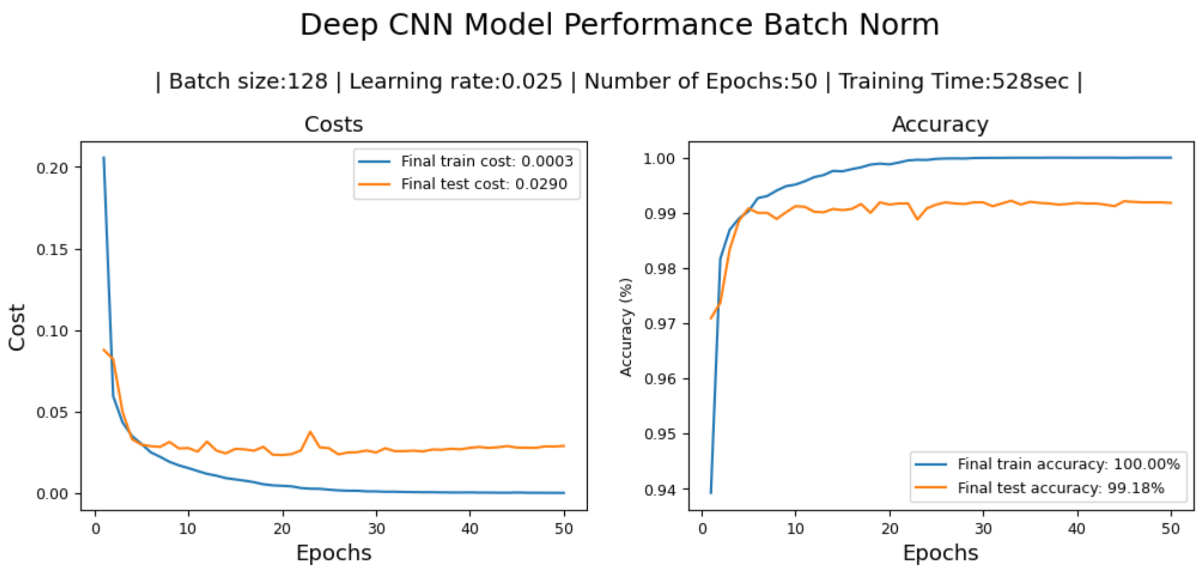


Figure 9: Cost and Accuracy using a residual network with batch normalization

Exercise 6b *Second variation*

For the second variation I decided to modify the parameters to the Adam optimizer from exercise 4.

I decided to decrease the learning rate from 0.001 to 0.0005 to try and get better results when near the end of training. I also changed the beta values from (0.9, 0.999) to (0.85,0.95) to see how much of an affect they would have. I used cross entropy loss as the loss function, the Adam optimizer with the above changes, 35 epochs, mini batch size of 35, and the default network architecture that was implemented in exercise 2. In Figure 10 we can see the results. Using these modifications resulted in slightly better performance with a test accuracy of 99.02%; however, this model appears to be overfitting since the test cost curve is slowly increasing.

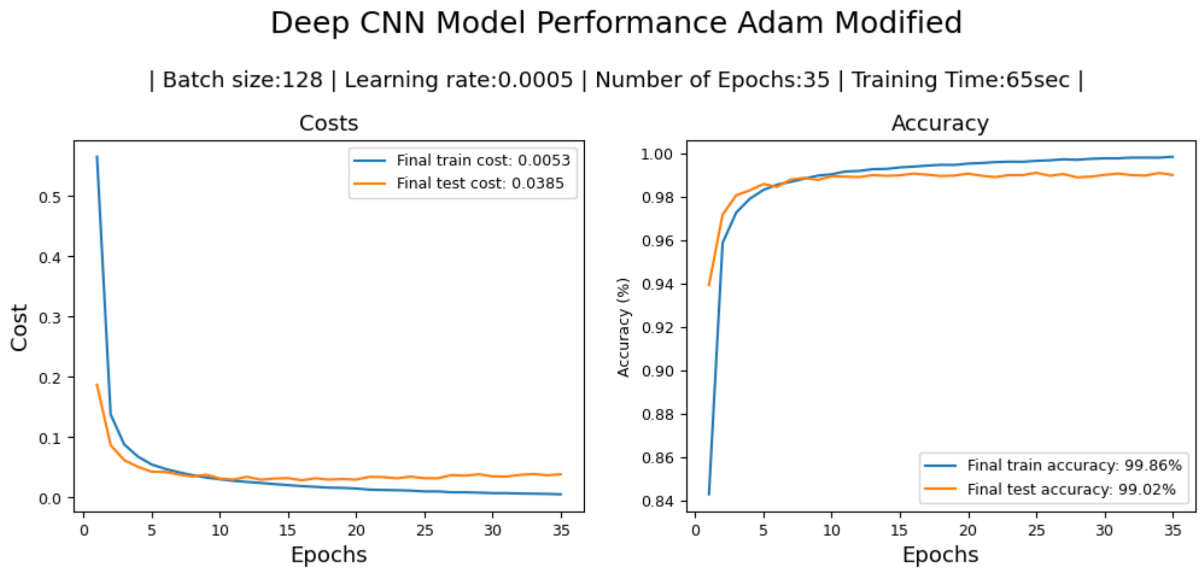


Figure 10: Cost and Accuracy using Adam Optimizer with modified parameters

Exercise 6c *Third variation*

For the third variation I decided to combine the deep residual network from 6a with the standard Adam optimizer to try and get the best of both worlds since these two methods have resulted in high test accuracies as shown in Figure 9 and Figure 6. I used the Adam optimizer, cross entropy loss, learning rate of 0.001, mini batch size of 128, trained for 25 epochs, and used the network architecture in 6a. the learning curve plot is shown is Figure 11. As you can see the test peformance is still good at 98.72% but not as good as it was in 6a. Here the learning rate may be too high causing these large jumps in the cost and accuracy functions. This model was on the verge of overfitting as seen in the test cost curve as it started to increase sharply.

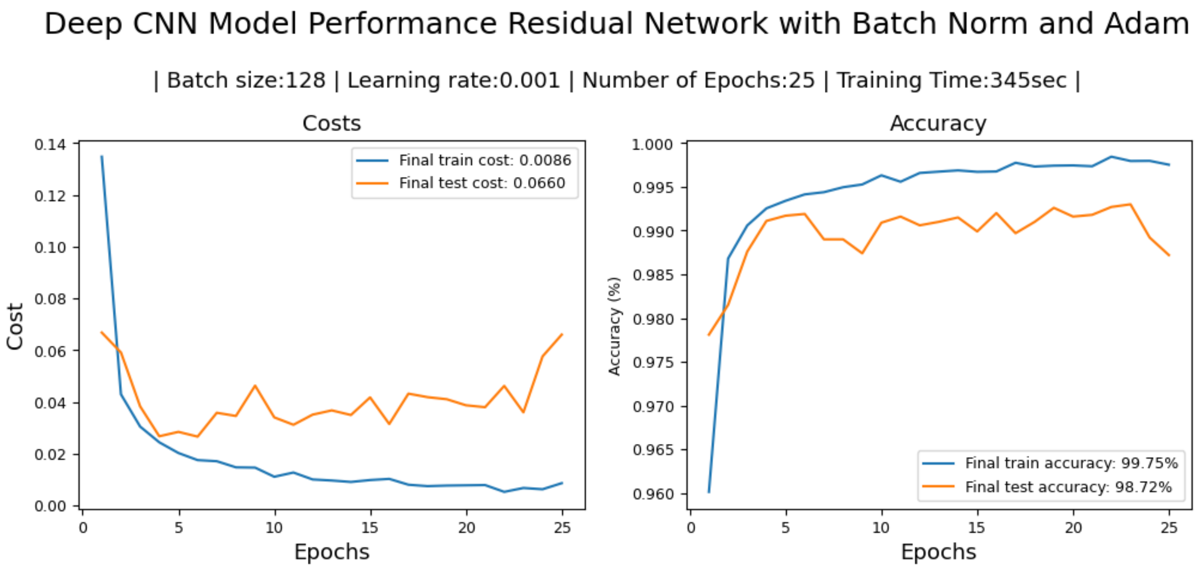


Figure 11: Cost and Accuracy using Adam Optimizer and Residual Network with batch normalization

Exercise 6 *Confusion Matrix*

The best performing model from exercise 6 was model 6a, the deep residual neural network with batch normalization. The confusion matrix on the test set is shown below in Figure 12.

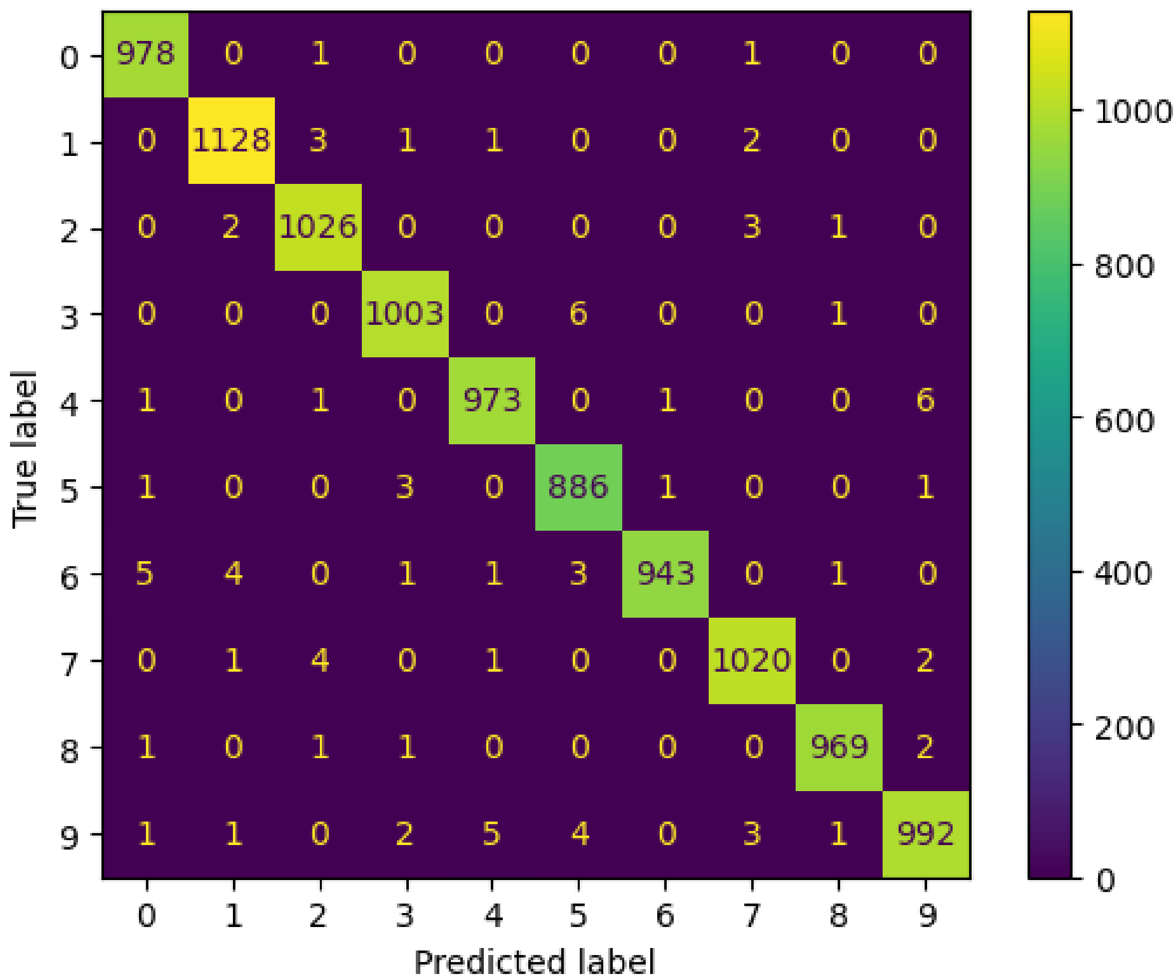
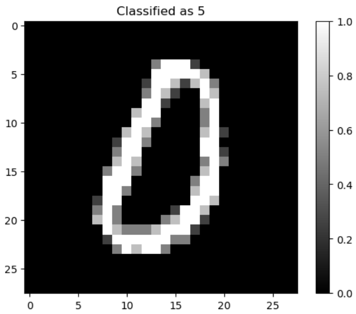


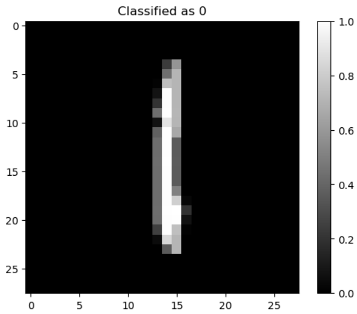
Figure 12: Confusion matrix for the best performing model (model 6a)

Exercise 6 *10 misclassifications*

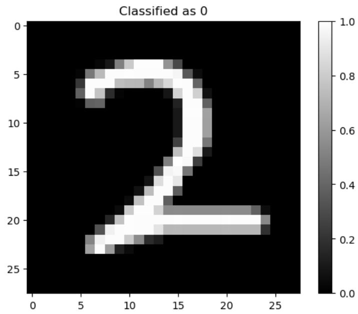
Below are 10 misclassifications and the digit that the model classified as shown in Figure 13. Misclassifications of 0 and 1 seem to be the most common in this sample probably due to the structure of 0 and 1 being represented in most other digits. That is circles or straight lines can be found in most other digits.



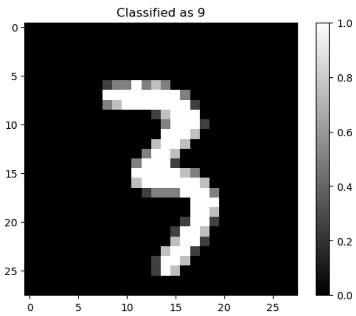
(a) 0 classified as 5



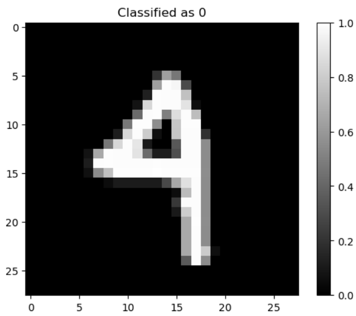
(b) 1 classified as 0



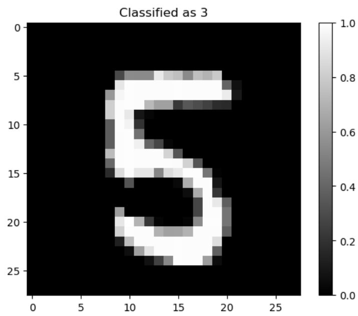
(c) 2 classified as 0



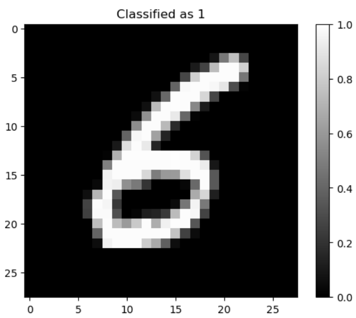
(d) 3 classified as 9



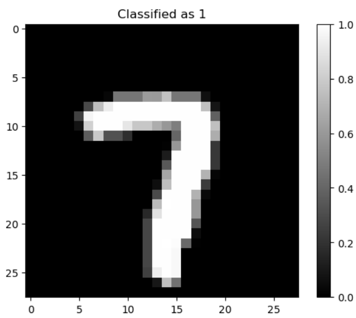
(e) 4 classified as 0



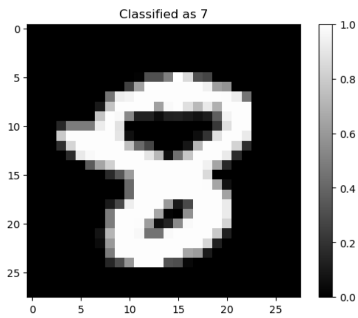
(f) 5 classified as 3



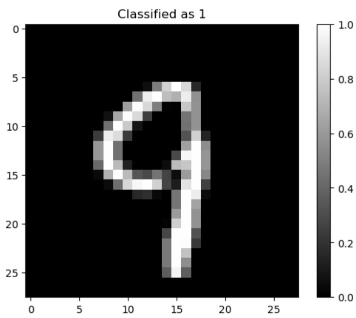
(g) 6 classified as 1



(h) 7 classified as 1



(i) 8 classified as 7



(j) 9 classified as 1

Figure 13: The written digit and what it was classified as

Use of generative AI

I only looked at the generative AI code when googling issues on the search results page; otherwise, I did not use generative AI.

References

[Pri23] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023. URL: http://udlbook.com.

A Code

```
#libraries we need
import numpy as np
import matplotlib.pyplot as plt
import load_mnist
import math
import time
import torch
from sklearn.utils import shuffle
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay

#Training test plot curve code that was provided in assignment 1 to get the plots
#modified it to contain the time and number of epochs
def training_curve_plot(title, train_costs, test_costs, train_accuracy, test_accuracy, batch_size, l
    lg=18
    md=13
    sm=9
    fig, axs = plt.subplots(1, 2, figsize=(12, 4))
    fig.suptitle(title, y=1.15, fontsize=lg)
    sub = f'| Batch size:{batch_size} | Learning rate:{learning_rate} | Number of Epochs:{epochs} | 
    fig.text(0.5, 0.99, sub, ha='center', fontsize=md)
    x = range(1, len(train_costs)+1)
    axs[0].plot(x, train_costs, label=f'Final train cost: {train_costs[-1]:.4f}')
    axs[0].plot(x, test_costs, label=f'Final test cost: {test_costs[-1]:.4f}')
    axs[0].set_title('Costs', fontsize=md)
    axs[0].set_xlabel('Epochs', fontsize=md)
    axs[0].set_ylabel('Cost', fontsize=md)
    axs[0].legend(fontsize=sm)
    axs[0].tick_params(axis='both', labelsize=sm)
    # Optionally use a logarithmic y-scale
    #axs[0].set_yscale('log')
    axs[1].plot(x, train_accuracy, label=f'Final train accuracy: {100*train_accuracy[-1]:.2f}%')
    axs[1].plot(x, test_accuracy, label=f'Final test accuracy: {100*test_accuracy[-1]:.2f}%')
    axs[1].set_title('Accuracy', fontsize=md)
    axs[1].set_xlabel('Epochs', fontsize=md)
    axs[1].set_ylabel('Accuracy (%)', fontsize=sm)
    axs[1].legend(fontsize=sm)
    axs[1].tick_params(axis='both', labelsize=sm)

#get the train and test data from the dataset
xtrain,ytrain,xtest,ytest = load_mnist.load_mnist()
#looking at the output
print("X Train shape", xtrain.shape)
print("Y Train shape", ytrain.shape)
print("X Test shape", xtest.shape)
print("Y Test shape", ytest.shape)

#converting to Tensors for easy PyTorch implementation
xtrain = torch.Tensor(xtrain).to("cpu")
ytrain = torch.Tensor(ytrain).to("cpu")
xtest = torch.Tensor(xtest).to("cpu")
ytest = torch.Tensor(ytest).to("cpu")

#first we want to put our data in a pytorch dataset so we can mini batch and enumerate through it la
train_dataset = torch.utils.data.TensorDataset(xtrain, ytrain)
```

```
test_dataset = torch.utils.data.TensorDataset(xtest, ytest)

#calculating the accuracy given outputs not softmaxed and labels one hot encoding.
def calculate_accuracy(outputs, labels):
    #don't need to softmax because the max value will be the max softmax we just pull the index to g
    _, output_index = torch.max(outputs,1)
    #get the index/ digit of the label
    _, label_index = torch.max(labels, 1)
    # return the number of correct matches and divide by the size to get accuracy
    return (output_index == label_index).sum().item()/labels.size(0)

#training loop function
def training_loop(train_loader, test_loader, num_epochs, model, loss_function, optimizer):
    #arrays for our plots
    training_loss = []
    training_accuracy = []
    test_loss = []
    test_accuracy =[]
    #Setting up the training loop
    print("Starting the Training Loop")
    for epoch in range(num_epochs):
        #keep the loss and accuracies after each mini batch
        batch_loss = []
        batch_accuracy = []
        #loop through a mini-batch on the same train loadear
        for batch_index, (data, label) in enumerate(train_loader):
            # Forward pass
            outputs = model(data)
            #evaluate the loss
            loss = loss_function(outputs, label)
            #append the loss to the batch loss
            batch_loss.append(loss.item())
            #calculate the accuracy based on the outputs (not softmaxed) and labels. Do outputs.data
            batch_accuracy.append(calculate_accuracy(outputs.data, label))

            # Backward pass setting gradients to zero
            optimizer.zero_grad()
            #calcualting gradients
            loss.backward()
            #updating parameters
            optimizer.step()

        #add to the training epoch accuracies and losses
        training_accuracy.append(np.average(batch_accuracy))
        training_loss.append(np.average(batch_loss))
        #get the test loss and accuracy
        #change mode
        model.eval()
        #so we don't accidentally change anything
        with torch.no_grad():
            #get the "batch" of the test data which is all of it
            for batch_index, (data, label) in enumerate(test_loader):
                #get our test predicitons
                test_predictions = model(data)
                #test loss and move to cpu so I can plot
                loss = loss_function(test_predictions, label).to("cpu")
                #append statistics
                test_loss.append(loss)
```

```python
                test_accuracy.append(calculate_accuracy(test_predictions.data, label))
        #back to training mode
        model.train()
        #printing
        print(f"Epoch: {epoch} done. Test loss {test_loss[epoch]}. Test accuracy {test_accuracy[epoc
    return training_loss, training_accuracy, test_loss, test_accuracy

# Here we make a neural network that is identical to the one in assignment 1
# so it will be a deep NN of 2 hidden layers, 50 nodes per layer
class Assignment1NN(torch.nn.Module):
    def __init__(self, input_size = 784, hidden_size = 50, output_size = 10):
        super().__init__()
        #First hiddent layer
        self.hidden1 = torch.nn.Linear(input_size, hidden_size)
        #ReLU activation function
        self.relu1 = torch.nn.ReLU()
        #second hidder layer
        self.hidden2 = torch.nn.Linear(hidden_size, hidden_size)
        #ReLU activation function
        self.relu2 = torch.nn.ReLU()
        #output layer
        self.output = torch.nn.Linear(hidden_size, output_size)
    #forward pass through the network
    def forward(self, x):
        #pass through first hidden layer
        x = self.hidden1(x)
        #activation function
        x = self.relu1(x)
        #hidden layer 2
        x = self.hidden2(x)
        #activation function
        x = self.relu2(x)
        #pass through the output layer
        x = self.output(x)
        return x

# setting the hyperparameters for exercise 1
input_size_1 = 784
num_classes_1 = 10
learning_rate_1 = 0.01
batch_size_1 = 128
num_epochs_1 = 150

#Making a dataloader for this specific NN which is a wrapper around the Dataset for easy use
train_loader_1 = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size_1, shuffle
#make the batch size for the test DataLoader the size of the dataset for evaluation.
test_loader_1 = torch.utils.data.DataLoader(dataset=test_dataset, batch_size = ytest.shape[0], shuff

#This is the Neural Network model
model_1 = Assignment1NN(input_size = input_size_1, hidden_size = 50, output_size = num_classes_1).to
#Our loss function will be cross entropy since we are getting a probability distribution
loss_1 = torch.nn.CrossEntropyLoss()
#Here we are going to use classic stochastic gradient descent without any special optimizations
optimizer_1 = torch.optim.SGD(model_1.parameters(), lr= learning_rate_1)
start_1 = time.time()
training_loss_1, training_accuracy_1, test_loss_1, test_accuracy_1 = training_loop(train_loader_1, t
num_epochs_1, model_1, loss_1,optimizer_1)
```

```
end_1 = time.time()
total_time = end_1 - start_1
#plotting
training_curve_plot("Deep NN (Pytorch) Model Performance", training_loss_1, test_loss_1, training_ac
128, 0.01, total_time, num_epochs_1)

sum_1 = 0
for param in model_1.parameters():
    sum_1 += param.numel()
print(sum_1)

# Exercise 2
# Implement a convolutional neural network

#will use local computer GPU to speed up training
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using device:", device)
print(torch.cuda.is_available())  # True if CUDA is available
print(torch.cuda.device_count())  # Number of GPUs available
print(torch.cuda.current_device())  # Current GPU index
print(torch.cuda.get_device_name(0))  # Name of the GPU

# since we are now working with a convolutional neural network we need to reshape the data to be a 2
# y stays the same
#reshape to N, Channels, height, width
xtrain_cnn = xtrain.reshape(60000, 1,28,28).to(device)
xtest_cnn = xtest.reshape(10000, 1,28,28).to(device)
ytrain_cnn = ytrain.to(device)
ytest_cnn = ytest.to(device)
#make our datasets so we can make data loaders
train_dataset_cnn = torch.utils.data.TensorDataset(xtrain_cnn, ytrain_cnn)
test_dataset_cnn = torch.utils.data.TensorDataset(xtest_cnn, ytest_cnn)

#make the CNN for exercise 2 according to the specifications in the assignment
class Exercise2CNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
        self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1,
        #non linearity
        self.relu1 = torch.nn.ReLU()
        #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
        self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
        self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, pa
        #non linearity
        self.relu2 = torch.nn.ReLU()
        #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
        self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
        self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, p
        #non linearity
        self.relu3 = torch.nn.ReLU()
        #output netwrok we have 32 channels and an image that is (7,7)
        self.output = torch.nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        #pass through the first convolution and relu and pooling layers
```

```
        x = self.pool1(self.relu1(self.conv1(x)))
        #pass through the second convolution and relu and pooling layers
        x = self.pool2(self.relu2(self.conv2(x)))
        #pass through the final convolution and relu
        x = self.relu3(self.conv3(x))
        #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
        x = torch.flatten(x, 1)
        #pass through our output layer
        x = self.output(x)
        return x

#want to set some hyperparameters
learning_rate_2 = 0.01
batch_size_2 = 128
num_epochs_2 = 100

#Making a dataloader for this specific CNN which is a wrapper around the Dataset for easy use
train_loader_cnn = torch.utils.data.DataLoader(dataset=train_dataset_cnn, batch_size=batch_size_1, s
#make the batch size for the test DataLoader the size of the dataset for evaluation.
test_loader_cnn = torch.utils.data.DataLoader(dataset=test_dataset_cnn, batch_size = ytest.shape[0],

#Make the CNN neural netowrk model
model_2 = Exercise2CNN().to(device)
#Our loss function will be cross entropy since we are getting a probability distribution
loss_2 = torch.nn.CrossEntropyLoss()
#Here we are going to use classic stochastic gradient descent without any special optimizations sinc
optimizer_2 = torch.optim.SGD(model_2.parameters(), lr= learning_rate_2)

#find the start time
start_2 = time.time()

#run the training loop
training_loss_2, training_accuracy_2, test_loss_2, test_accuracy_2 = training_loop(train_loader_cnn,
num_epochs_2, model_2, loss_2, optimizer_2)

#end time and get the total time
end_2 = time.time()
total_time = end_2 - start_2
#plotting
training_curve_plot("Deep CNN Model Performance", training_loss_2, test_loss_2, training_accuracy_2,
batch_size_2, learning_rate_2, total_time, num_epochs_2)

sum_2 = 0
for param in model_2.parameters():
    sum_2 += param.numel()
print(sum_2)

# Exercise 3

#make the CNN for exercise 3 according to the specifications in the assignment
#which is almost identical to exercise 2
class Exercise3CNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
        self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1,
        #non linearity
        self.relu1 = torch.nn.ReLU()
```

```
        #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
        self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
        self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, pad
        #non linearity
        self.relu2 = torch.nn.ReLU()
        #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
        self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
        self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, pa
        #non linearity
        self.relu3 = torch.nn.ReLU()
        #output netwrok we have 32 channels and an image that is (7,7)
        self.output = torch.nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        #pass through the first convolution and relu and pooling layers
        x = self.relu1(self.pool1(self.conv1(x)))
        #pass through the second convolution and relu and pooling layers
        x = self.relu2(self.pool2(self.conv2(x)))
        #pass through the final convolution and relu
        x = self.relu3(self.conv3(x))
        #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
        x = torch.flatten(x, 1)
        #pass through our output layer
        x = self.output(x)
        return x

#want to set some hyperparameters
learning_rate_3 = 0.01
batch_size_3 = 128
num_epochs_3 = 100

#Make the CNN neural netowrk model
model_3 = Exercise3CNN().to(device)
#Our loss function will be cross entropy since we are getting a probability distribution
loss_3 = torch.nn.CrossEntropyLoss()
#Here we are going to use classic stochastic gradient descent without any special optimizations sinc
optimizer_3 = torch.optim.SGD(model_3.parameters(), lr= learning_rate_3)

#find the start time
start_3 = time.time()

#run the training loop
training_loss_3, training_accuracy_3, test_loss_3, test_accuracy_3 = training_loop(train_loader_cnn,
num_epochs_3, model_3, loss_3, optimizer_3)

#end time and get the total time
end_3 = time.time()
total_time = end_3 - start_3
#plotting
training_curve_plot("Deep CNN Model Performance Pooling then ReLU", training_loss_3, test_loss_3, tra
batch_size_3, learning_rate_3, total_time, num_epochs_3)

#make the CNN for exercise 3b according to the specifications in the assignment
#which is almost identical to exercise 2 but we use tanh
class Exercise3bCNN(torch.nn.Module):
    def __init__(self):
```

```
        super().__init__()
        #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
        self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1,
        #non linearity
        self.tanh1 = torch.nn.Tanh()
        #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
        self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
        self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, pad
        #non linearity
        self.tanh2 = torch.nn.Tanh()
        #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
        self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
        self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, pa
        #non linearity
        self.tanh3 = torch.nn.Tanh()
        #output netwrok we have 32 channels and an image that is (7,7)
        self.output = torch.nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        #pass through the first convolution and relu and pooling layers
        x = self.tanh1(self.pool1(self.conv1(x)))
        #pass through the second convolution and relu and pooling layers
        x = self.tanh2(self.pool2(self.conv2(x)))
        #pass through the final convolution and relu
        x = self.tanh3(self.conv3(x))
        #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
        x = torch.flatten(x, 1)
        #pass through our output layer
        x = self.output(x)
        return x

#want to set some hyperparameters
learning_rate_3b = 0.01
batch_size_3b = 128
num_epochs_3b = 100

#Make the CNN neural netowrk model
model_3b = Exercise3bCNN().to(device)
#Our loss function will be cross entropy since we are getting a probability distribution
loss_3b = torch.nn.CrossEntropyLoss()
#Here we are going to use classic stochastic gradient descent without any special optimizations sinc
optimizer_3b = torch.optim.SGD(model_3b.parameters(), lr= learning_rate_3b)

#find the start time
start_3b = time.time()

#run the training loop
training_loss_3b, training_accuracy_3b, test_loss_3b, test_accuracy_3b = training_loop(train_loader_
num_epochs_3b, model_3b, loss_3b, optimizer_3b)

#end time and get the total time
end_3b = time.time()
total_time = end_3b - start_3b
#plotting
training_curve_plot("Deep CNN Model Performance Pooling then TanH", training_loss_3b, test_loss_3b, t
batch_size_3b, learning_rate_3b, total_time, num_epochs_3b)
```

```

# Exercise 4

#want to set some hyperparameters
learning_rate_4 = 0.001
batch_size_4 = 128
num_epochs_4 = 100

#Make the CNN neural netowrk model
model_4 = Exercise2CNN().to(device)
#Our loss function will be cross entropy since we are getting a probability distribution
loss_4 = torch.nn.CrossEntropyLoss()
#Here we are going to use classic stochastic gradient descent without any special optimizations sinc
optimizer_4 = torch.optim.Adam(model_4.parameters(), lr= learning_rate_4)

#find the start time
start_4 = time.time()

#run the training loop
training_loss_4, training_accuracy_4, test_loss_4, test_accuracy_4 = training_loop(train_loader_cnn,
num_epochs_4, model_4, loss_4, optimizer_4)

#end time and get the total time
end_4 = time.time()
total_time = end_4 - start_4
#plotting
training_curve_plot("Deep CNN Model Performance Adam Optimizer", training_loss_4, test_loss_4, train
batch_size_4, learning_rate_4, total_time, num_epochs_4)

# Exercise 5
#make the CNN residual network for exercise 5 according to the specifications in the assignment
#resdiual architecture for the 8 channels to 8 channels
class FirstKernelResidual(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
        self.conv = torch.nn.Conv2d(in_channels = 8, out_channels = 8, kernel_size = 3, stride = 1, p
        #here are the relu and pooling layers
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        #modeling the books residual connection from section 11.2 figure 11.5b
        # pass it through ReLU
        temp = self.relu(x)
        #then we pass it through our convolutional layer where we relu then pool
        temp = self.conv(temp)
        #add in our residual connection
        x = x + temp
        return x

#residual architecture for the 16 channels to 16 channels
class SecondKernelResidual(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
        self.conv = torch.nn.Conv2d(in_channels = 16, out_channels = 16, kernel_size = 3, stride = 1
        #here are the relu and pooling layers
        self.relu = torch.nn.ReLU()
```

```python

    def forward(self, x):
        #modeling the books residual connection from section 11.2 figure 11.5b
        #pass through relu
        temp = self.relu(x)
        #then we pass it through our convolutional layer where we relu then pool
        temp = self.conv(temp)
        #add in our residual connection
        x = x + temp
        return x

#residual connections for 32 channels to 32 channels
class ThirdKernelResidual(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
        self.conv = torch.nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size = 3, stride = 1
        #here are the relu and pooling layers
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        #modeling the books residual connection from section 11.2 figure 11.5b
        #we pass it through ReLU
        temp = self.relu(x)
        #then we pass it through our convolutional layer where we relu then pool
        temp = self.conv(temp)
        #add in our residual connection
        x = x + temp
        return x


class Exercise5CNN(torch.nn.Module):
    def __init__(self, first = 3,second = 3, third = 3):
        super().__init__()
        #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
        self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1,
        #adding many residual layers in this case default 3 more convolutions
        self.layer1 = torch.nn.ModuleList([FirstKernelResidual() for _ in range(first)])
        #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
        self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
        self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, pa
        #adding many residual layers in this case default 3 more convolutions at the 16 channel leng
        self.layer2 = torch.nn.ModuleList([SecondKernelResidual() for _ in range(second)])
        #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
        self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
        self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, p
        #adding many residual layers in this case default 3 more convolutions at the 32 channel leng
        self.layer3 = torch.nn.ModuleList([ThirdKernelResidual() for _ in range(third)])
        #non linearity
        self.relu3 = torch.nn.ReLU()
        #output neturok we have 32 channels and an image that is (7,7)
        self.output = torch.nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        #pass through the first convolution
        #we don't need to relu because we pass it through several residual layers that will handel t
```

```
        x = self.conv1(x)
        #pass through several residual layers
        for l in self.layer1:
            x = l(x)
        #pooling at the end of residual connections because pool will decrease our image size
        x = self.pool1(x)
        #pass through the second convolution to get to the next channel size of 16 and smaller image
        x = self.conv2(x)
        #pass through several residual layers
        for l in self.layer2:
            x = l(x)
        #pooling to reduce size for the third convolutional layer
        x = self.pool2(x)
        #pass through the final convolution
        x = self.conv3(x)
        #pass through more residual convolutions then
        for l in self.layer3:
            x = l(x)
        #final relu
        x = self.relu3(x)
        #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
        x = torch.flatten(x, 1)
        #pass through our output layer
        x = self.output(x)
        return x

#want to set some hyperparameters
learning_rate_5 = 0.01
batch_size_5 = 128
num_epochs_5 = 50

#Make the CNN neural netowrk model
model_5 = Exercise5CNN().to(device)
#Our loss function will be cross entropy since we are getting a probability distribution
loss_5 = torch.nn.CrossEntropyLoss()
#Here we are going to use classic stochastic gradient descent without any special optimizations sinc
optimizer_5 = torch.optim.SGD(model_5.parameters(), lr= learning_rate_5)

#find the start time
start_5 = time.time()

#run the training loop
training_loss_5, training_accuracy_5, test_loss_5, test_accuracy_5 = training_loop(train_loader_cnn,
num_epochs_5, model_5, loss_5, optimizer_5)

#end time and get the total time
end_5 = time.time()
total_time = end_5 - start_5
#plotting
training_curve_plot("Deep CNN Model Performance 9 residual layers", training_loss_5, test_loss_5, tra
batch_size_5, learning_rate_5, total_time, num_epochs_5)

#5b deep neural network no residuals
#make the CNN residual network for exercise 5 according to the specifications in the assignment

#resdiual architecture for the 8 channels to 8 channels
class FirstKernelResidual5b(torch.nn.Module):
    def __init__(self):
```

```
        super().__init__()
        #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
        self.conv = torch.nn.Conv2d(in_channels = 8, out_channels = 8, kernel_size = 3, stride = 1, p
        #here are the relu and pooling layers
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        #modeling the books residual connection from section 11.2 figure 11.5b
        # pass it through ReLU
        x = self.relu(x)
        #then we pass it through our convolutional layer where we relu then pool
        x = self.conv(x)
        return x

#residual architecture for the 16 channels to 16 channels
class SecondKernelResidual5b(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
        self.conv = torch.nn.Conv2d(in_channels = 16, out_channels = 16, kernel_size = 3, stride = 1
        #here are the relu and pooling layers
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        #modeling the books residual connection from section 11.2 figure 11.5b
        #pass through relu
        temp = self.relu(x)
        #then we pass it through our convolutional layer where we relu then pool
        x = self.conv(x)
        return x

#residual connections for 32 channels to 32 channels
class ThirdKernelResidual5b(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
        self.conv = torch.nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size = 3, stride = 1
        #here are the relu and pooling layers
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        #modeling the books residual connection from section 11.2 figure 11.5b
        #we pass it through ReLU
        x = self.relu(x)
        #then we pass it through our convolutional layer where we relu then pool
        x = self.conv(x)
        return x


class Exercise5bCNN(torch.nn.Module):
    def __init__(self, first = 3,second = 3, third = 3):
        super().__init__()
        #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
        self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1,
        #adding many residual layers in this case default 3 more convolutions
        self.layer1 = torch.nn.ModuleList([FirstKernelResidual5b() for _ in range(first)])
        #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
        self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
```

```
        #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
        self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, pad
        #adding many residual layers in this case default 3 more convolutions at the 16 channel leng
        self.layer2 = torch.nn.ModuleList([SecondKernelResidual5b() for _ in range(second)])
        #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
        self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
        self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, pa
        #adding many residual layers in this case default 3 more convolutions at the 32 channel leng
        self.layer3 = torch.nn.ModuleList([ThirdKernelResidual5b() for _ in range(third)])
        #non linearity
        self.relu3 = torch.nn.ReLU()
        #output netwrok we have 32 channels and an image that is (7,7)
        self.output = torch.nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        #pass through the first convolution
        #we don't need to relu because we pass it through several residual layers that will handel t
        x = self.conv1(x)
        #pass through several residual layers
        for l in self.layer1:
            x = l(x)
        #pooling at the end of residual connections because pool will decrease our image size
        x = self.pool1(x)
        #pass through the second convolution to get to the next channel size of 16 and smaller image
        x = self.conv2(x)
        #pass through several residual layers
        for l in self.layer2:
            x = l(x)
        #pooling to reduce size for the third convolutional layer
        x = self.pool2(x)
        #pass through the final convolution
        x = self.conv3(x)
        #pass through more residual convolutions then
        for l in self.layer3:
            x = l(x)
        #final relu
        x = self.relu3(x)
        #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
        x = torch.flatten(x, 1)
        #pass through our output layer
        x = self.output(x)
        return x

#want to set some hyperparameters
learning_rate_5b = 0.01
batch_size_5b = 128
num_epochs_5b = 50

#Make the CNN neural netowrk model
model_5b = Exercise5bCNN().to(device)
#Our loss function will be cross entropy since we are getting a probability distribution
loss_5b = torch.nn.CrossEntropyLoss()
#Here we are going to use classic stochastic gradient descent without any special optimizations sinc
optimizer_5b = torch.optim.SGD(model_5b.parameters(), lr= learning_rate_5b)

#find the start time
start_5b = time.time()
```

```

#run the training loop
training_loss_5b, training_accuracy_5b, test_loss_5b, test_accuracy_5b = training_loop(train_loader_
num_epochs_5b, model_5b, loss_5b, optimizer_5b)

#end time and get the total time
end_5b = time.time()
total_time = end_5b - start_5b
#plotting
training_curve_plot("Deep CNN Model Performance 9 extra layers no residuals", training_loss_5b, test_
batch_size_5b, learning_rate_5b, total_time, num_epochs_5b)

# Exercise 6a
# Adding batch normalization to residual layers
#make the CNN residual network for exercise 6 with batch normalization added which is a regularizati

#resdiual architecture for the 8 channels to 8 channels
class FirstKernelResidual6a(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
        self.conv = torch.nn.Conv2d(in_channels = 8, out_channels = 8, kernel_size = 3, stride = 1, p
        #also want to batch normalize the input
        #takes input (N,C,H,W) where C is channels I think so in this case 8
        self.norm = torch.nn.BatchNorm2d(8)
        #here are the relu and pooling layers
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        #modeling the books residual connection from section 11.2 figure 11.5b and 11.6 for batch no
        #first we batch normalize
        temp = self.norm(x)
        #then we pass it through ReLU
        temp = self.relu(temp)
        #then we pass it through our convolutional layer where we relu then pool
        temp = self.conv(temp)
        #add in our residual connection
        x = x + temp
        return x

#residual architecture for the 16 channels to 16 channels
class SecondKernelResidual6a(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
        self.conv = torch.nn.Conv2d(in_channels = 16, out_channels = 16, kernel_size = 3, stride = 1
        #also want to batch normalize the input
        #takes input (N,C,H,W) where C is channels I think so in this case 8
        self.norm = torch.nn.BatchNorm2d(16)
        #here are the relu and pooling layers
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        #modeling the books residual connection from section 11.2 figure 11.5b
        #first we batch normalize
        temp = self.norm(x)
        #then we pass it through ReLU
        temp = self.relu(temp)
```

```
        #then we pass it through our convolutional layer where we relu then pool
        temp = self.conv(temp)
        #add in our residual connection
        x = x + temp
        return x

#residual connections for 32 channels to 32 channels
class ThirdKernelResidual6a(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
        self.conv = torch.nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size = 3, stride = 1
        #also want to batch normalize the input
        #takes input (N,C,H,W) where C is channels I think so in this case 8
        self.norm = torch.nn.BatchNorm2d(32)
        #here are the relu and pooling layers
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        #modeling the books residual connection from section 11.2 figure 11.5b
        #first we batch normalize
        temp = self.norm(x)
        #then we pass it through ReLU
        temp = self.relu(temp)
        #then we pass it through our convolutional layer where we relu then pool
        temp = self.conv(temp)
        #add in our residual connection
        x = x + temp
        return x


class Exercise6aCNN(torch.nn.Module):
    def __init__(self, first = 3,second = 3, third = 3):
        super().__init__()
        #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
        self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1,
        #adding many residual layers in this case default 3 more convolutions
        self.layer1 = torch.nn.ModuleList([FirstKernelResidual6a() for _ in range(first)])
        #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
        self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
        self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, pad
        #adding many residual layers in this case default 3 more convolutions at the 16 channel leng
        self.layer2 = torch.nn.ModuleList([SecondKernelResidual6a() for _ in range(second)])
        #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
        self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
        # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
        self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, pa
        #adding many residual layers in this case default 3 more convolutions at the 32 channel leng
        self.layer3 = torch.nn.ModuleList([ThirdKernelResidual6a() for _ in range(third)])
        #non linearity
        self.relu3 = torch.nn.ReLU()
        #output netwrok we have 32 channels and an image that is (7,7)
        self.output = torch.nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        #pass through the first convolution
        #we don't need to relu because we pass it through several residual layers that will handel t
```

```
        x = self.conv1(x)
        #pass through several residual layers
        for l in self.layer1:
            x = l(x)
        #pooling at the end of residual connections because pool will decrease our image size
        x = self.pool1(x)
        #pass through the second convolution to get to the next channel size of 16 and smaller image
        x = self.conv2(x)
        #pass through several residual layers
        for l in self.layer2:
            x = l(x)
        #pooling to reduce size for the third convolutional layer
        x = self.pool2(x)
        #pass through the final convolution
        x = self.conv3(x)
        #pass through more residual convolutions then
        for l in self.layer3:
            x = l(x)
        #final relu
        x = self.relu3(x)
        #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
        x = torch.flatten(x, 1)
        #pass through our output layer
        x = self.output(x)
        return x

#want to set some hyperparameters
learning_rate_6a = 0.025
batch_size_6a = 128
num_epochs_6a = 50

#Make the CNN neural netowrk model
model_6a = Exercise6aCNN().to(device)
#Our loss function will be cross entropy since we are getting a probability distribution
loss_6a = torch.nn.CrossEntropyLoss()
#Here we are going to use classic stochastic gradient descent without any special optimizations sinc
optimizer_6a = torch.optim.SGD(model_6a.parameters(), lr= learning_rate_6a)

#find the start time
start_6a = time.time()

#run the training loop
training_loss_6a, training_accuracy_6a, test_loss_6a, test_accuracy_6a = training_loop(train_loader_
num_epochs_6a, model_6a, loss_6a, optimizer_6a)

#end time and get the total time
end_6a = time.time()
total_time = end_6a - start_6a
#plotting
training_curve_plot("Deep CNN Model Performance Batch Norm", training_loss_6a, test_loss_6a, trainin
batch_size_6a, learning_rate_6a, total_time, num_epochs_6a)

# Exercise 6b
# Modifying Adam parameters
#want to set some hyperparameters
learning_rate_6b = 0.0005
batch_size_6b = 128
num_epochs_6b = 35
```

```
#Make the CNN neural netowrk model
model_6b = Exercise2CNN().to(device)
#Our loss function will be cross entropy since we are getting a probability distribution
loss_6b = torch.nn.CrossEntropyLoss()
#Here we are going to use classic stochastic gradient descent without any special optimizations sinc
optimizer_6b = torch.optim.Adam(model_6b.parameters(), lr= learning_rate_6b, betas = (0.85,0.95))

#find the start time
start_6b = time.time()

#run the training loop
training_loss_6b, training_accuracy_6b, test_loss_6b, test_accuracy_6b = training_loop(train_loader_
num_epochs_6b, model_6b, loss_6b, optimizer_6b)

#end time and get the total time
end_6b = time.time()
total_time = end_6b - start_6b
#plotting
training_curve_plot("Deep CNN Model Performance Adam Modified", training_loss_6b, test_loss_6b, trai
batch_size_6b, learning_rate_6b, total_time, num_epochs_6b)

# Exercise 6c
# Residual Network with Batch Norm and Adam

#want to set some hyperparameters
learning_rate_6c = 0.001
batch_size_6c = 128
num_epochs_6c = 25

#Make the CNN neural netowrk model
model_6c = Exercise6aCNN().to(device)
#Our loss function will be cross entropy since we are getting a probability distribution
loss_6c = torch.nn.CrossEntropyLoss()
#Here we are going to use classic stochastic gradient descent without any special optimizations sinc
optimizer_6c = torch.optim.Adam(model_6c.parameters(), lr= learning_rate_6c)

#find the start time
start_6c = time.time()

#run the training loop
training_loss_6c, training_accuracy_6c, test_loss_6c, test_accuracy_6c = training_loop(train_loader_
num_epochs_6c, model_6c, loss_6c, optimizer_6c)

#end time and get the total time
end_6c = time.time()
total_time = end_6c - start_6c
#plotting
training_curve_plot("Deep CNN Model Performance Residual Network with Batch Norm and Adam", training
test_loss_6c, training_accuracy_6c, test_accuracy_6c, batch_size_6c, learning_rate_6c, total_time, nu
```