

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 its 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.

Deep NN (Numpy) Model Performance

| Batch size:128 | Learning rate:0.01 | Number of Epochs:150 | Training Time:139sec |

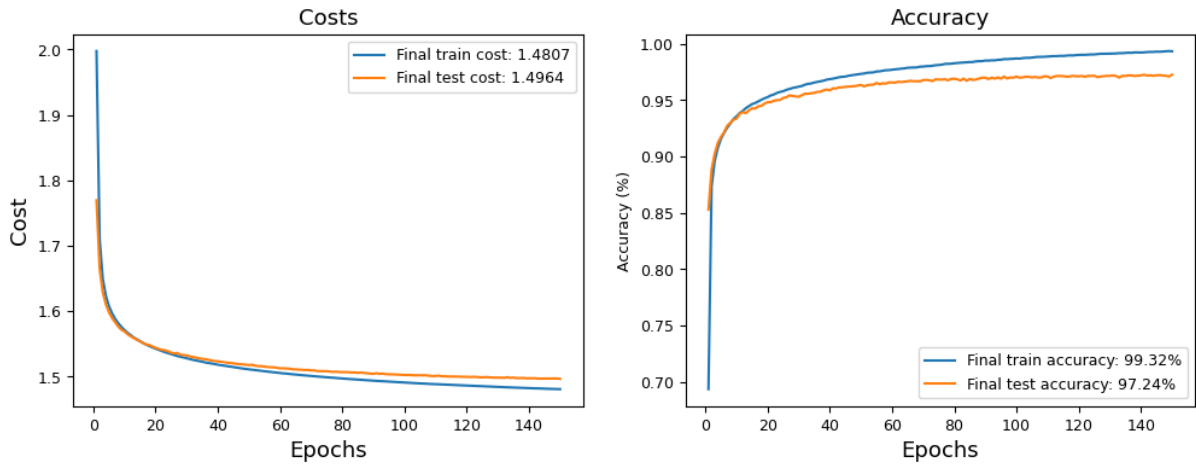


Figure 1: Cost and Accuracy for my Numpy implementation of a deep Neural Network

Deep NN (Pytorch) Model Performance

| Batch size:128 | Learning rate:0.01 | Number of Epochs:150 | Training Time:199sec |

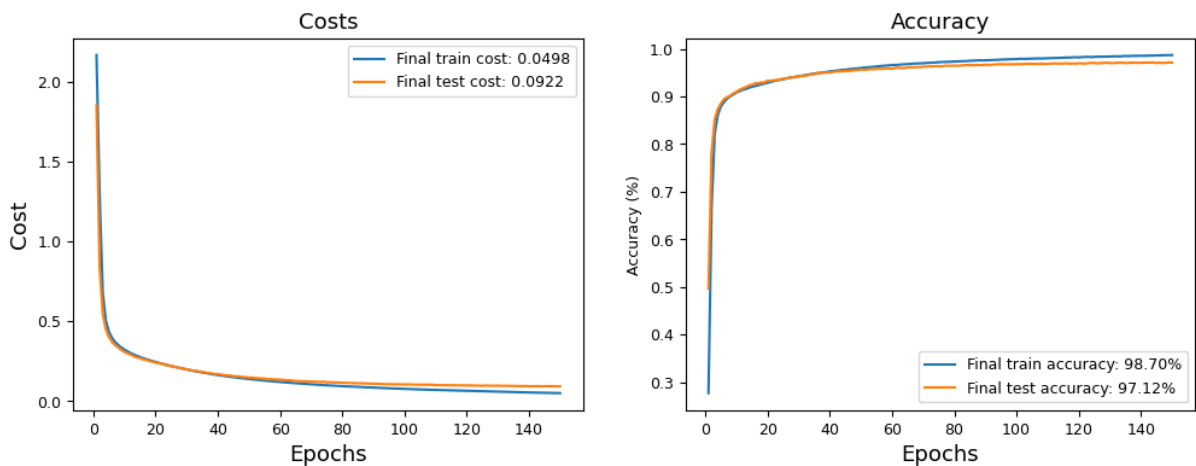


Figure 2: Cost and Accuracy for my PyTorch implementation of a deep Neural Network

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%

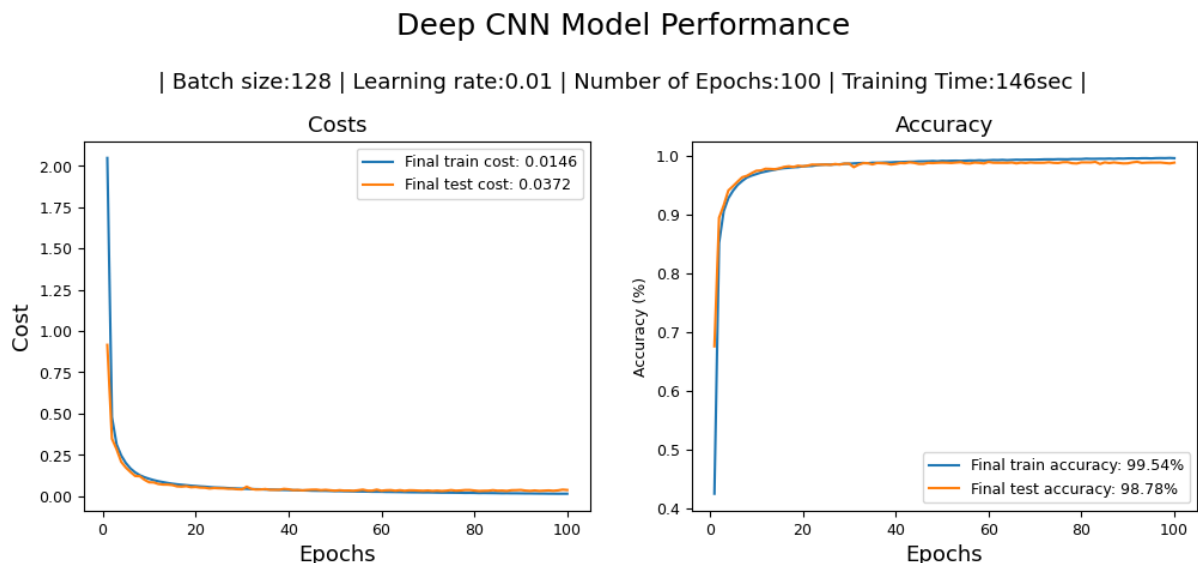


Figure 3: Cost and Accuracy for my implementation of a deep Convolutional Neural Network

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

Deep CNN Model Performance Pooling then ReLU

| Batch size:128 | Learning rate:0.01 | Number of Epochs:100 | Training Time:204sec |

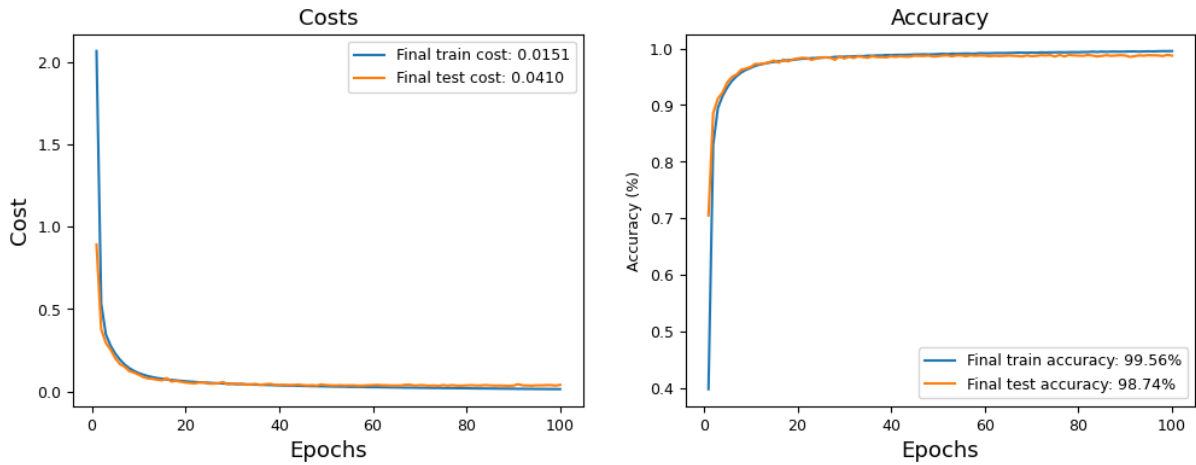


Figure 4: Cost and Accuracy for pooling being executed before ReLU

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.

Deep CNN Model Performance Pooling then TanH

| Batch size:128 | Learning rate:0.01 | Number of Epochs:100 | Training Time:162sec |

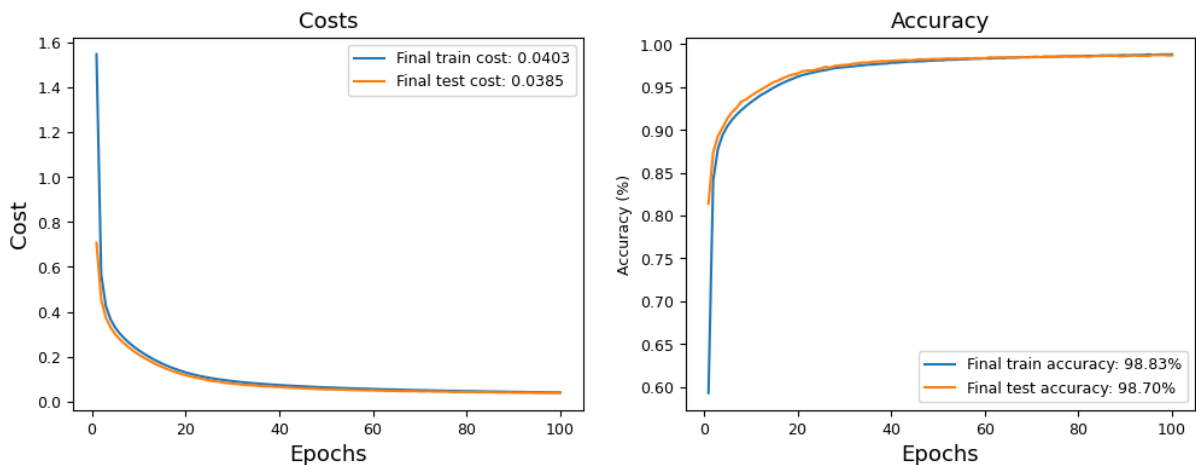


Figure 5: Cost and Accuracy for pooling being executed before TanH

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 $[0, \infty]$, 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.

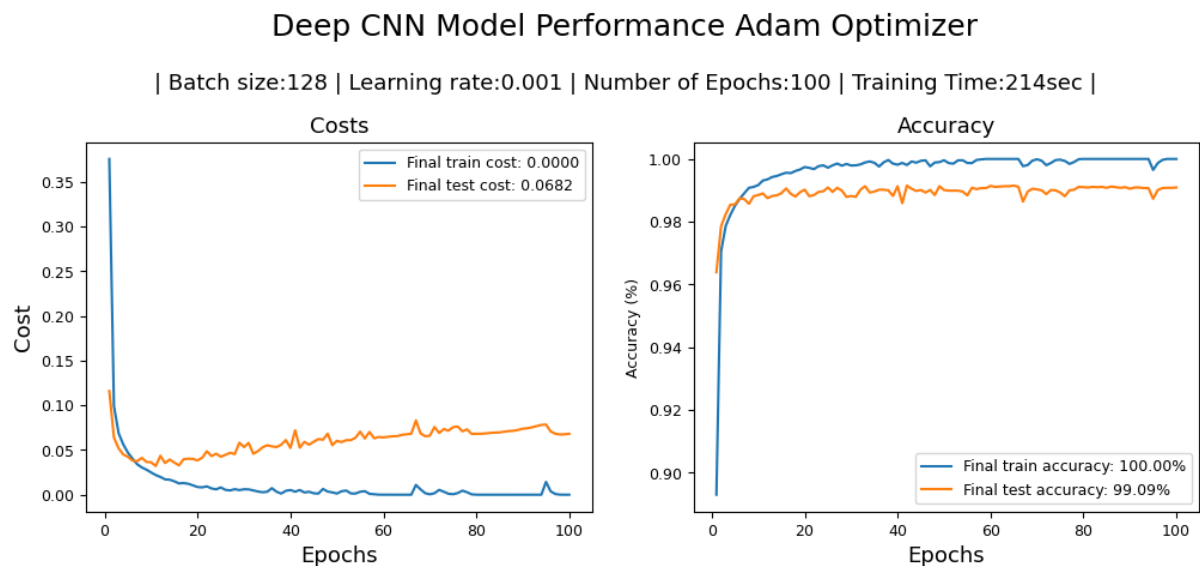


Figure 6: Cost and Accuracy using the Adam optimizer

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 differ-

ence. 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.

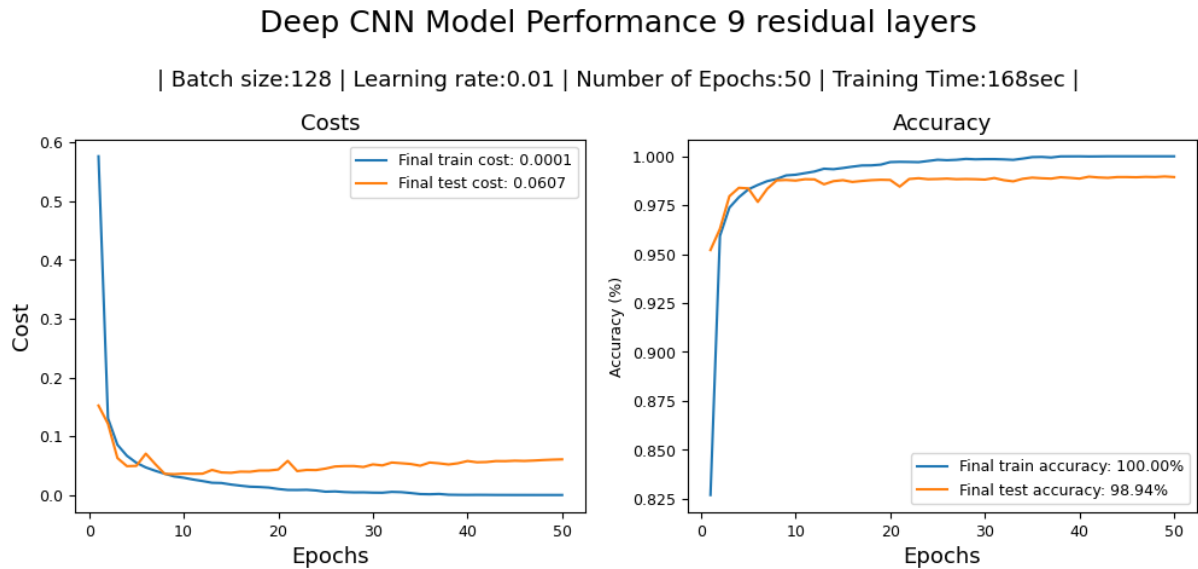


Figure 7: Cost and Accuracy using the 9 extra residual layers and using the SGD optimizer

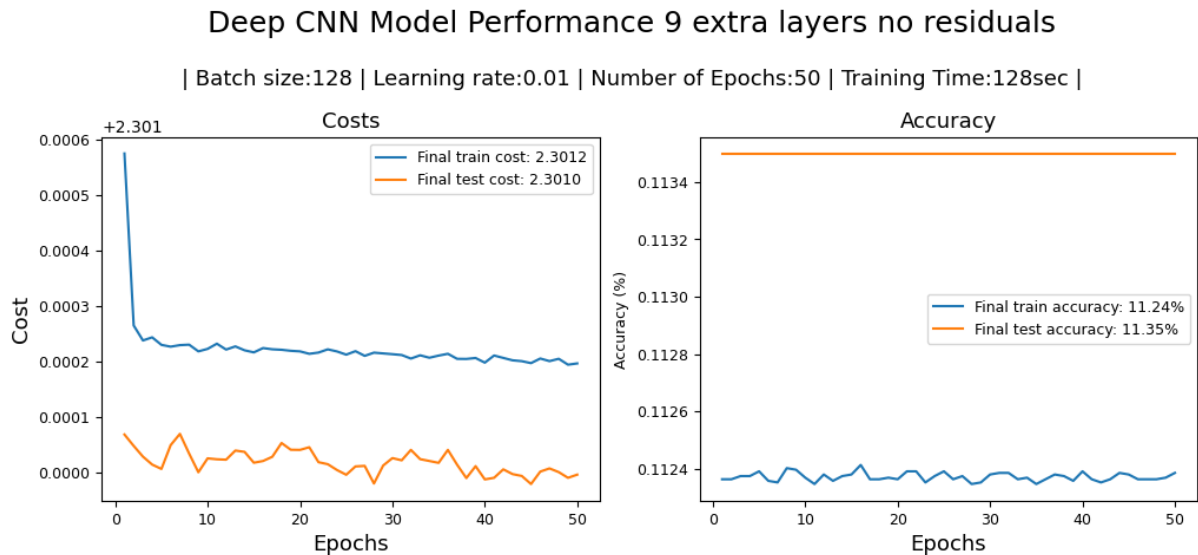


Figure 8: Cost and Accuracy using the 9 extra layers and using the SGD optimizer. No residual connections

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.

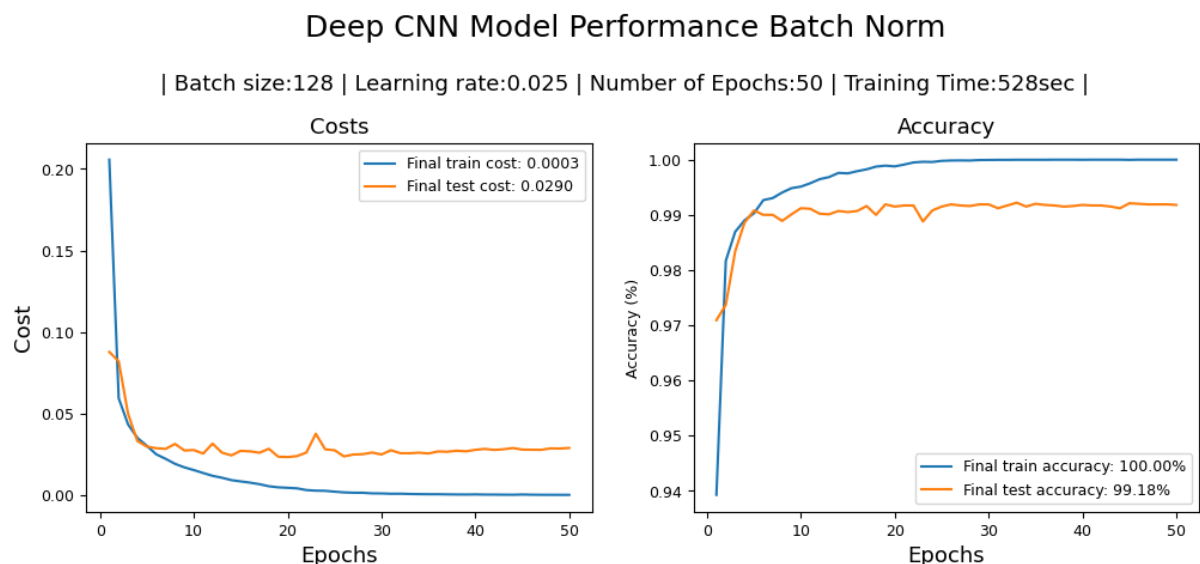


Figure 9: Cost and Accuracy using a residual network with batch normalization

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.

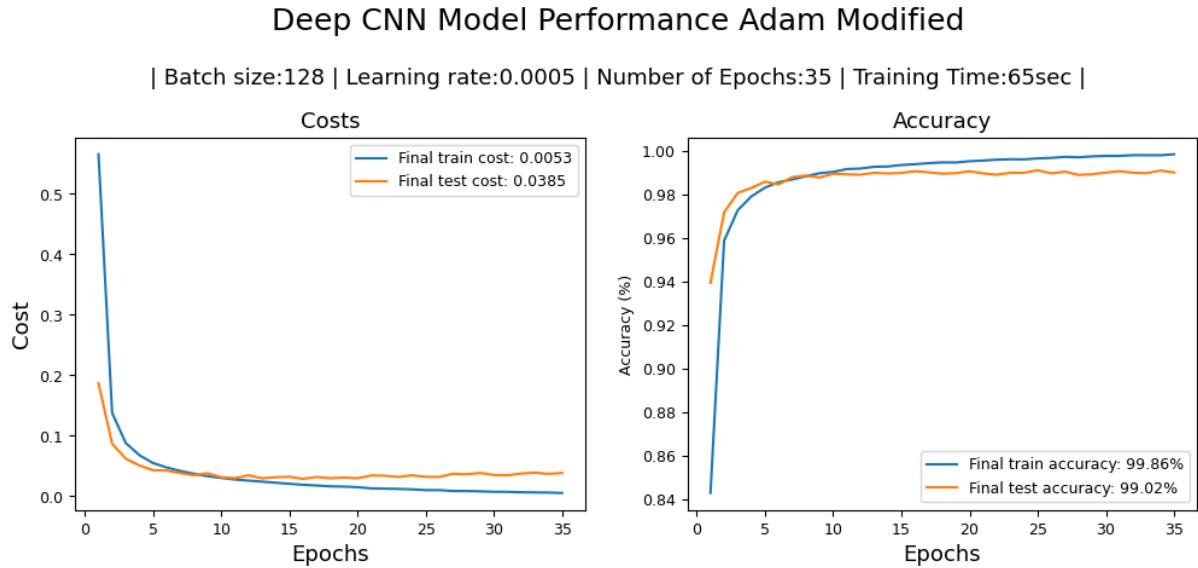


Figure 10: Cost and Accuracy using Adam Optimizer with modified parameters

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 performance 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.

Deep CNN Model Performance Residual Network with Batch Norm and Adam

| Batch size:128 | Learning rate:0.001 | Number of Epochs:25 | Training Time:345sec |

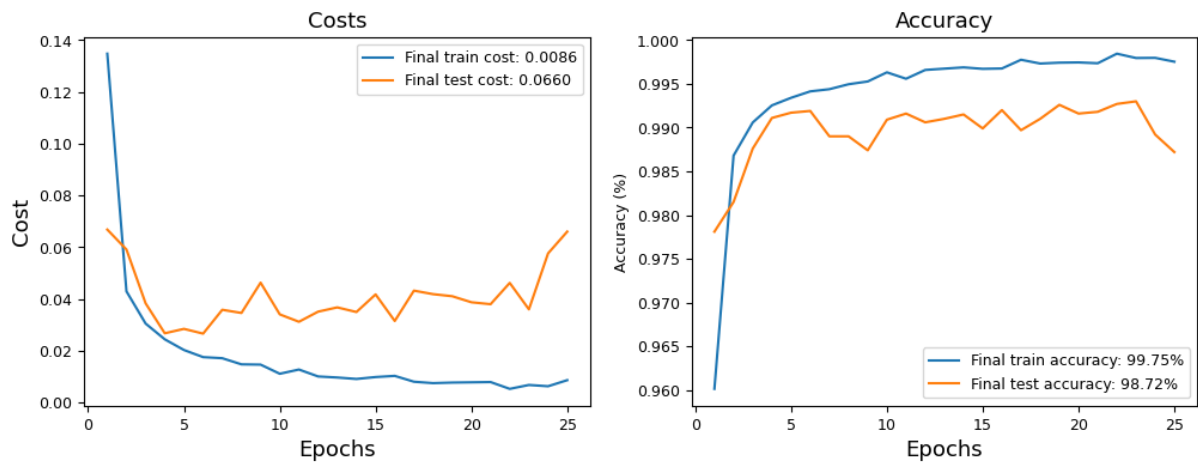


Figure 11: Cost and Accuracy using Adam Optimizer and Residual Network with batch normalization

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.

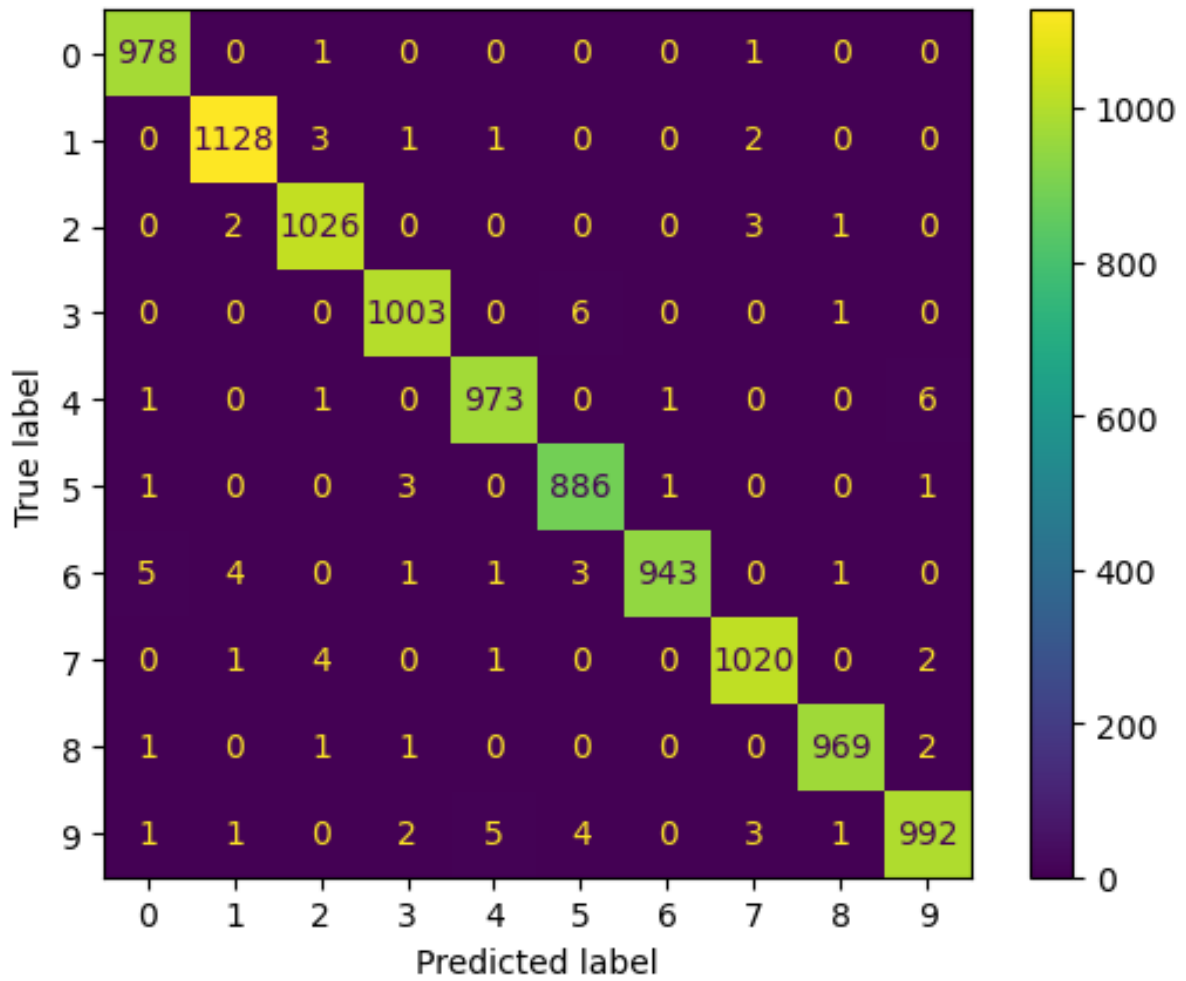


Figure 12: Confusion matrix for the best performing model (model 6a)

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.

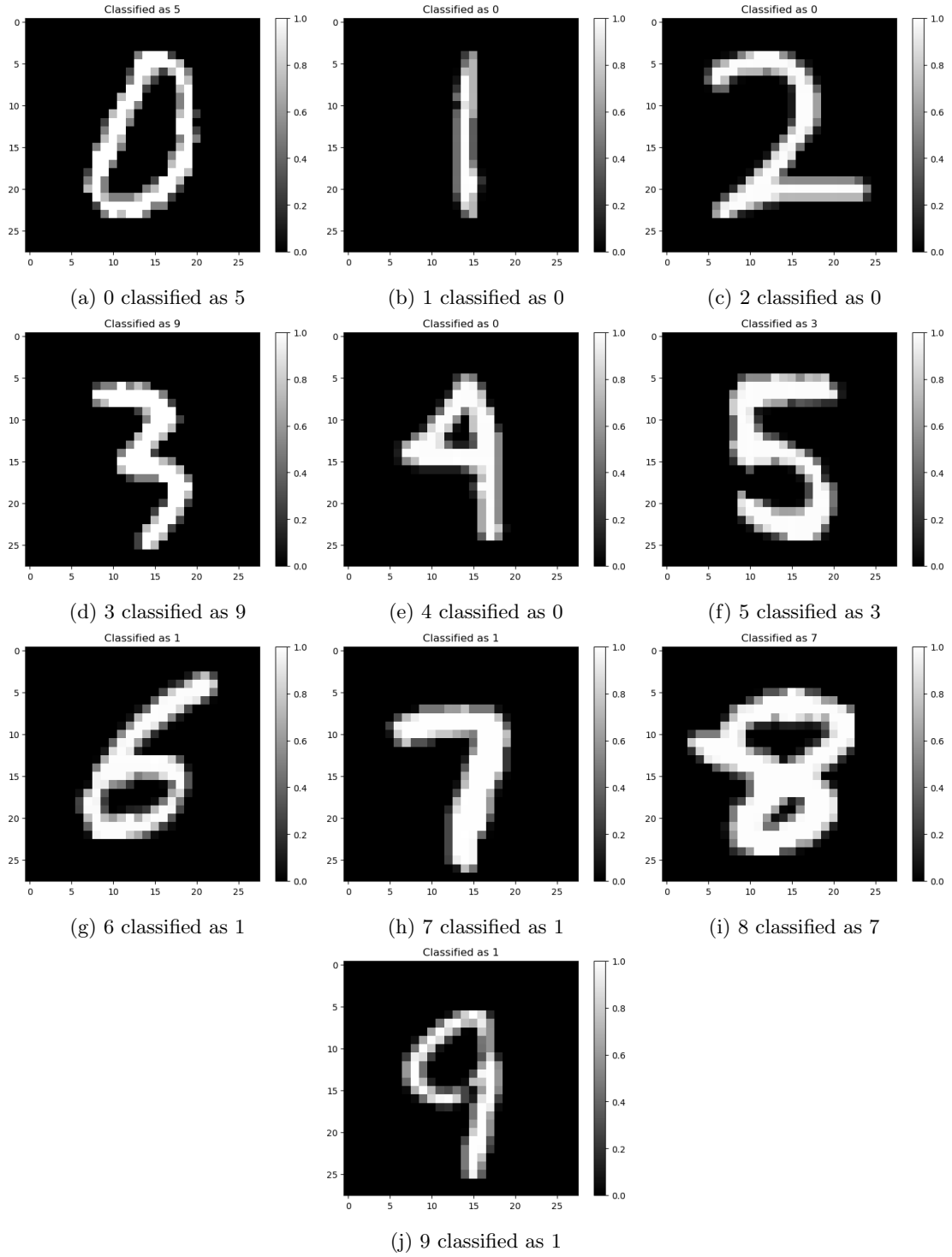


Figure 13: The written digit and what it was classified as

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

```
1  #libraries we need
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import load_mnist
5  import math
6  import time
7  import torch
8  from sklearn.utils import shuffle
9  from sklearn.metrics import confusion_matrix
10 from sklearn.metrics import ConfusionMatrixDisplay
11
12 #Training test plot curve code that was provided in assignment 1 to get the plots
13 #modified it to contain the time and number of epochs
14 def training_curve_plot(title, train_costs, test_costs, train_accuracy, test_accuracy, batch_size, l
15     lg=18
16     md=13
17     sm=9
18     fig, axs = plt.subplots(1, 2, figsize=(12, 4))
19     fig.suptitle(title, y=1.15, fontsize=lg)
20     sub = f'| Batch size:{batch_size} | Learning rate:{learning_rate} | Number of Epochs:{epochs} | '
21     fig.text(0.5, 0.99, sub, ha='center', fontsize=md)
22     x = range(1, len(train_costs)+1)
23     axs[0].plot(x, train_costs, label=f'Final train cost: {train_costs[-1]:.4f}')
24     axs[0].plot(x, test_costs, label=f'Final test cost: {test_costs[-1]:.4f}')
25     axs[0].set_title('Costs', fontsize=md)
26     axs[0].set_xlabel('Epochs', fontsize=md)
27     axs[0].set_ylabel('Cost', fontsize=md)
28     axs[0].legend(fontsize=sm)
29     axs[0].tick_params(axis='both', labelsize=sm)
30     # Optionally use a logarithmic y-scale
31     #axs[0].set_yscale('log')
32     axs[1].plot(x, train_accuracy, label=f'Final train accuracy: {100*train_accuracy[-1]:.2f}%')
33     axs[1].plot(x, test_accuracy, label=f'Final test accuracy: {100*test_accuracy[-1]:.2f}%')
34     axs[1].set_title('Accuracy', fontsize=md)
35     axs[1].set_xlabel('Epochs', fontsize=md)
36     axs[1].set_ylabel('Accuracy (%)', fontsize=sm)
37     axs[1].legend(fontsize=sm)
38     axs[1].tick_params(axis='both', labelsize=sm)
39
40 #get the train and test data from the dataset
41 xtrain,ytrain,xtest,ytest = load_mnist.load_mnist()
42 #looking at the output
43 print("X Train shape", xtrain.shape)
44 print("Y Train shape", ytrain.shape)
45 print("X Test shape", xtest.shape)
46 print("Y Test shape", ytest.shape)
47
48 #converting to Tensors for easy PyTorch implementation
49 xtrain = torch.Tensor(xtrain).to("cpu")
50 ytrain = torch.Tensor(ytrain).to("cpu")
51 xtest = torch.Tensor(xtest).to("cpu")
52 ytest = torch.Tensor(ytest).to("cpu")
53
54 #first we want to put our data in a pytorch dataset so we can mini batch and enumerate through it la
55 train_dataset = torch.utils.data.TensorDataset(xtrain, ytrain)
```

```

56 test_dataset = torch.utils.data.TensorDataset(xtest, ytest)
57
58 #calculating the accuracy given outputs not softmaxed and labels one hot encoding.
59 def calculate_accuracy(outputs, labels):
60     #don't need to softmax because the max value will be the max softmax we just pull the index to g
61     _, output_index = torch.max(outputs,1)
62     #get the index/ digit of the label
63     _, label_index = torch.max(labels, 1)
64     # return the number of correct matches and divide by the size to get accuracy
65     return (output_index == label_index).sum().item()/labels.size(0)
66
67 #training loop function
68 def training_loop(train_loader, test_loader, num_epochs, model, loss_function, optimizer):
69     #arrays for our plots
70     training_loss = []
71     training_accuracy = []
72     test_loss = []
73     test_accuracy = []
74     #Setting up the training loop
75     print("Starting the Training Loop")
76     for epoch in range(num_epochs):
77         #keep the loss and accuracies after each mini batch
78         batch_loss = []
79         batch_accuracy = []
80         #loop through a mini-batch on the same train loadear
81         for batch_index, (data, label) in enumerate(train_loader):
82             # Forward pass
83             outputs = model(data)
84             #evaluate the loss
85             loss = loss_function(outputs, label)
86             #append the loss to the batch loss
87             batch_loss.append(loss.item())
88             #calculate the accuracy based on the outputs (not softmaxed) and labels. Do outputs.data
89             batch_accuracy.append(calculate_accuracy(outputs.data, label))
90
91             # Backward pass setting gradients to zero
92             optimizer.zero_grad()
93             #calculating gradients
94             loss.backward()
95             #updating parameters
96             optimizer.step()
97
98     #add to the training epoch accuracies and losses
99     training_accuracy.append(np.average(batch_accuracy))
100    training_loss.append(np.average(batch_loss))
101    #get the test loss and accuracy
102    #change mode
103    model.eval()
104    #so we don't accidentally change anything
105    with torch.no_grad():
106        #get the "batch" of the test data which is all of it
107        for batch_index, (data, label) in enumerate(test_loader):
108            #get our test predicitions
109            test_predictions = model(data)
110            #test loss and move to cpu so I can plot
111            loss = loss_function(test_predictions, label).to("cpu")
112            #append statistics
113            test_loss.append(loss)

```

```

114         test_accuracy.append(calculate_accuracy(test_predictions.data, label))
115         #back to training mode
116         model.train()
117         #printing
118         print(f"Epoch: {epoch} done. Test loss {test_loss[epoch]}. Test accuracy {test_accuracy[epoch]}")
119     return training_loss, training_accuracy, test_loss, test_accuracy
120
121     # Here we make a neural network that is identical to the one in assignment 1
122     # so it will be a deep NN of 2 hidden layers, 50 nodes per layer
123     class Assignment1NN(torch.nn.Module):
124         def __init__(self, input_size = 784, hidden_size = 50, output_size = 10):
125             super().__init__()
126             #First hidden layer
127             self.hidden1 = torch.nn.Linear(input_size, hidden_size)
128             #ReLU activation function
129             self.relu1 = torch.nn.ReLU()
130             #second hidden layer
131             self.hidden2 = torch.nn.Linear(hidden_size, hidden_size)
132             #ReLU activation function
133             self.relu2 = torch.nn.ReLU()
134             #output layer
135             self.output = torch.nn.Linear(hidden_size, output_size)
136             #forward pass through the network
137         def forward(self, x):
138             #pass through first hidden layer
139             x = self.hidden1(x)
140             #activation function
141             x = self.relu1(x)
142             #hidden layer 2
143             x = self.hidden2(x)
144             #activation function
145             x = self.relu2(x)
146             #pass through the output layer
147             x = self.output(x)
148             return x
149
150     # setting the hyperparameters for exercise 1
151     input_size_1 = 784
152     num_classes_1 = 10
153     learning_rate_1 = 0.01
154     batch_size_1 = 128
155     num_epochs_1 = 150
156
157     #Making a dataloader for this specific NN which is a wrapper around the Dataset for easy use
158     train_loader_1 = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size_1, shuffle=True)
159     #make the batch size for the test DataLoader the size of the dataset for evaluation.
160     test_loader_1 = torch.utils.data.DataLoader(dataset=test_dataset, batch_size = ytest.shape[0], shuffle=False)
161
162     #This is the Neural Network model
163     model_1 = Assignment1NN(input_size = input_size_1, hidden_size = 50, output_size = num_classes_1).to(device)
164     #Our loss function will be cross entropy since we are getting a probability distribution
165     loss_1 = torch.nn.CrossEntropyLoss()
166     #Here we are going to use classic stochastic gradient descent without any special optimizations
167     optimizer_1 = torch.optim.SGD(model_1.parameters(), lr= learning_rate_1)
168     start_1 = time.time()
169     training_loss_1, training_accuracy_1, test_loss_1, test_accuracy_1 = training_loop(train_loader_1, test_loader_1,
170     num_epochs_1, model_1, loss_1,optimizer_1)
171

```



```

172 end_1 = time.time()
173 total_time = end_1 - start_1
174 #plotting
175 training_curve_plot("Deep NN (Pytorch) Model Performance", training_loss_1, test_loss_1, training_acc_1,
176 128, 0.01, total_time, num_epochs_1)
177
178 sum_1 = 0
179 for param in model_1.parameters():
180     sum_1 += param.numel()
181 print(sum_1)
182
183 # Exercise 2
184 # Implement a convolutional neural network
185
186 #will use local computer GPU to speed up training
187 device = "cuda" if torch.cuda.is_available() else "cpu"
188 print("Using device:", device)
189 print(torch.cuda.is_available()) # True if CUDA is available
190 print(torch.cuda.device_count()) # Number of GPUs available
191 print(torch.cuda.current_device()) # Current GPU index
192 print(torch.cuda.get_device_name(0)) # Name of the GPU
193
194 # since we are now working with a convolutional neural network we need to reshape the data to be a 2D array
195 # y stays the same
196 #reshape to N, Channels, height, width
197 xtrain_cnn = xtrain.reshape(60000, 1,28,28).to(device)
198 xtest_cnn = xtest.reshape(10000, 1,28,28).to(device)
199 ytrain_cnn = ytrain.to(device)
200 ytest_cnn = ytest.to(device)
201 #make our datasets so we can make data loaders
202 train_dataset_cnn = torch.utils.data.TensorDataset(xtrain_cnn, ytrain_cnn)
203 test_dataset_cnn = torch.utils.data.TensorDataset(xtest_cnn, ytest_cnn)
204
205 #make the CNN for exercise 2 according to the specifications in the assignment
206 class Exercise2CNN(torch.nn.Module):
207     def __init__(self):
208         super().__init__()
209         #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
210         self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1, padding = 1)
211         #non linearity
212         self.relu1 = torch.nn.ReLU()
213         #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
214         self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
215         #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
216         self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, padding = 1)
217         #non linearity
218         self.relu2 = torch.nn.ReLU()
219         #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
220         self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
221         # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
222         self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, padding = 1)
223         #non linearity
224         self.relu3 = torch.nn.ReLU()
225         #output network we have 32 channels and an image that is (7,7)
226         self.output = torch.nn.Linear(32 * 7 * 7, 10)
227
228     def forward(self, x):
229         #pass through the first convolution and relu and pooling layers

```

```

230         x = self.pool1(self.relu1(self.conv1(x)))
231         #pass through the second convolution and relu and pooling layers
232         x = self.pool2(self.relu2(self.conv2(x)))
233         #pass through the final convolution and relu
234         x = self.relu3(self.conv3(x))
235         #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
236         x = torch.flatten(x, 1)
237         #pass through our output layer
238         x = self.output(x)
239         return x
240
241     #want to set some hyperparameters
242     learning_rate_2 = 0.01
243     batch_size_2 = 128
244     num_epochs_2 = 100
245
246     #Making a dataloader for this specific CNN which is a wrapper around the Dataset for easy use
247     train_loader_cnn = torch.utils.data.DataLoader(dataset=train_dataset_cnn, batch_size=batch_size_1, s
248     #make the batch size for the test DataLoader the size of the dataset for evaluation.
249     test_loader_cnn = torch.utils.data.DataLoader(dataset=test_dataset_cnn, batch_size = ytest.shape[0],
250
251     #Make the CNN neural netowrk model
252     model_2 = Exercise2CNN().to(device)
253     #Our loss function will be cross entropy since we are getting a probability distribution
254     loss_2 = torch.nn.CrossEntropyLoss()
255     #Here we are going to use classic stochastic gradient descent without any special optimizations sinc
256     optimizer_2 = torch.optim.SGD(model_2.parameters(), lr= learning_rate_2)
257
258     #find the start time
259     start_2 = time.time()
260
261     #run the training loop
262     training_loss_2, training_accuracy_2, test_loss_2, test_accuracy_2 = training_loop(train_loader_cnn,
263     num_epochs_2, model_2, loss_2, optimizer_2)
264
265     #end time and get the total time
266     end_2 = time.time()
267     total_time = end_2 - start_2
268     #plotting
269     training_curve_plot("Deep CNN Model Performance", training_loss_2, test_loss_2, training_accuracy_2,
270     batch_size_2, learning_rate_2, total_time, num_epochs_2)
271
272     sum_2 = 0
273     for param in model_2.parameters():
274         sum_2 += param.numel()
275     print(sum_2)
276
277     # Exercise 3
278
279     #make the CNN for exercise 3 according to the specifications in the assignment
280     #which is almost identical to exercise 2
281     class Exercise3CNN(torch.nn.Module):
282         def __init__(self):
283             super().__init__()
284             #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
285             self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1,
286             #non linearity
287             self.relu1 = torch.nn.ReLU()

```

```

288         #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
289         self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
290         #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
291         self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, padding= 1)
292         #non linearity
293         self.relu2 = torch.nn.ReLU()
294         #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
295         self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
296         # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
297         self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, padding= 1)
298         #non linearity
299         self.relu3 = torch.nn.ReLU()
300         #output network we have 32 channels and an image that is (7,7)
301         self.output = torch.nn.Linear(32 * 7 * 7, 10)
302
303     def forward(self, x):
304         #pass through the first convolution and relu and pooling layers
305         x = self.relu1(self.pool1(self.conv1(x)))
306         #pass through the second convolution and relu and pooling layers
307         x = self.relu2(self.pool2(self.conv2(x)))
308         #pass through the final convolution and relu
309         x = self.relu3(self.conv3(x))
310         #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
311         x = torch.flatten(x, 1)
312         #pass through our output layer
313         x = self.output(x)
314         return x
315
316     #want to set some hyperparameters
317     learning_rate_3 = 0.01
318     batch_size_3 = 128
319     num_epochs_3 = 100
320
321     #Make the CNN neural network model
322     model_3 = Exercise3CNN().to(device)
323     #Our loss function will be cross entropy since we are getting a probability distribution
324     loss_3 = torch.nn.CrossEntropyLoss()
325     #Here we are going to use classic stochastic gradient descent without any special optimizations since
326     optimizer_3 = torch.optim.SGD(model_3.parameters(), lr= learning_rate_3)
327
328     #find the start time
329     start_3 = time.time()
330
331     #run the training loop
332     training_loss_3, training_accuracy_3, test_loss_3, test_accuracy_3 = training_loop(train_loader_cnn,
333     num_epochs_3, model_3, loss_3, optimizer_3)
334
335     #end time and get the total time
336     end_3 = time.time()
337     total_time = end_3 - start_3
338     #plotting
339     training_curve_plot("Deep CNN Model Performance Pooling then ReLU", training_loss_3, test_loss_3, training_accuracy_3,
340     batch_size_3, learning_rate_3, total_time, num_epochs_3)
341
342     #make the CNN for exercise 3b according to the specifications in the assignment
343     #which is almost identical to exercise 2 but we use tanh
344     class Exercise3bCNN(torch.nn.Module):
345         def __init__(self):

```

```

346         super().__init__()
347         #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
348         self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1, padding = 1)
349         #non linearity
350         self.tanh1 = torch.nn.Tanh()
351         #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
352         self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
353         #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
354         self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, padding = 1)
355         #non linearity
356         self.tanh2 = torch.nn.Tanh()
357         #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
358         self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
359         # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
360         self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, padding = 1)
361         #non linearity
362         self.tanh3 = torch.nn.Tanh()
363         #output network we have 32 channels and an image that is (7,7)
364         self.output = torch.nn.Linear(32 * 7 * 7, 10)
365
366     def forward(self, x):
367         #pass through the first convolution and relu and pooling layers
368         x = self.tanh1(self.pool1(self.conv1(x)))
369         #pass through the second convolution and relu and pooling layers
370         x = self.tanh2(self.pool2(self.conv2(x)))
371         #pass through the final convolution and relu
372         x = self.tanh3(self.conv3(x))
373         #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
374         x = torch.flatten(x, 1)
375         #pass through our output layer
376         x = self.output(x)
377         return x
378
379     #want to set some hyperparameters
380     learning_rate_3b = 0.01
381     batch_size_3b = 128
382     num_epochs_3b = 100
383
384     #Make the CNN neural network model
385     model_3b = Exercise3bCNN().to(device)
386     #Our loss function will be cross entropy since we are getting a probability distribution
387     loss_3b = torch.nn.CrossEntropyLoss()
388     #Here we are going to use classic stochastic gradient descent without any special optimizations since
389     optimizer_3b = torch.optim.SGD(model_3b.parameters(), lr= learning_rate_3b)
390
391     #find the start time
392     start_3b = time.time()
393
394     #run the training loop
395     training_loss_3b, training_accuracy_3b, test_loss_3b, test_accuracy_3b = training_loop(train_loader, model_3b,
396     num_epochs_3b, model_3b, loss_3b, optimizer_3b)
397
398     #end time and get the total time
399     end_3b = time.time()
400     total_time = end_3b - start_3b
401     #plotting
402     training_curve_plot("Deep CNN Model Performance Pooling then TanH", training_loss_3b, test_loss_3b, test_accuracy_3b,
403     batch_size_3b, learning_rate_3b, total_time, num_epochs_3b)

```

```

404
405 # Exercise 4
406
407 #want to set some hyperparameters
408 learning_rate_4 = 0.001
409 batch_size_4 = 128
410 num_epochs_4 = 100
411
412 #Make the CNN neural netowrk model
413 model_4 = Exercise2CNN().to(device)
414 #Our loss function will be cross entropy since we are getting a probability distribution
415 loss_4 = torch.nn.CrossEntropyLoss()
416 #Here we are going to use classic stochastic gradient descent without any special optimizations sinc
417 optimizer_4 = torch.optim.Adam(model_4.parameters(), lr= learning_rate_4)
418
419 #find the start time
420 start_4 = time.time()
421
422 #run the training loop
423 training_loss_4, training_accuracy_4, test_loss_4, test_accuracy_4 = training_loop(train_loader_cnn,
424 num_epochs_4, model_4, loss_4, optimizer_4)
425
426 #end time and get the total time
427 end_4 = time.time()
428 total_time = end_4 - start_4
429 #plotting
430 training_curve_plot("Deep CNN Model Performance Adam Optimizer", training_loss_4, test_loss_4, train
431 batch_size_4, learning_rate_4, total_time, num_epochs_4)
432
433 # Exercise 5
434 #make the CNN residual network for exercise 5 according to the specifications in the assignment
435 #residual architecture for the 8 channels to 8 channels
436 class FirstKernelResidual(torch.nn.Module):
437     def __init__(self):
438         super().__init__()
439         #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
440         self.conv = torch.nn.Conv2d(in_channels = 8, out_channels = 8, kernel_size = 3, stride = 1,
441         #here are the relu and pooling layers
442         self.relu = torch.nn.ReLU()
443
444     def forward(self, x):
445         #modeling the books residual connection from section 11.2 figure 11.5b
446         # pass it through ReLU
447         temp = self.relu(x)
448         #then we pass it through our convolutional layer where we relu then pool
449         temp = self.conv(temp)
450         #add in our residual connection
451         x = x + temp
452         return x
453
454 #residual architecture for the 16 channels to 16 channels
455 class SecondKernelResidual(torch.nn.Module):
456     def __init__(self):
457         super().__init__()
458         #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
459         self.conv = torch.nn.Conv2d(in_channels = 16, out_channels = 16, kernel_size = 3, stride = 1
460         #here are the relu and pooling layers
461         self.relu = torch.nn.ReLU()

```

462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519

```
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, padding = 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, padding = 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, padding = 1)  
        #adding many residual layers in this case default 3 more convolutions at the 16 channel length  
        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, padding = 1)  
        #adding many residual layers in this case default 3 more convolutions at the 32 channel length  
        self.layer3 = torch.nn.ModuleList([ThirdKernelResidual() for _ in range(third)])  
        #non linearity  
        self.relu3 = torch.nn.ReLU()  
        #output network 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
```

```

520     x = self.conv1(x)
521     #pass through several residual layers
522     for l in self.layer1:
523         x = l(x)
524     #pooling at the end of residual connections because pool will decrease our image size
525     x = self.pool1(x)
526     #pass through the second convolution to get to the next channel size of 16 and smaller image
527     x = self.conv2(x)
528     #pass through several residual layers
529     for l in self.layer2:
530         x = l(x)
531     #pooling to reduce size for the third convolutional layer
532     x = self.pool2(x)
533     #pass through the final convolution
534     x = self.conv3(x)
535     #pass through more residual convolutions then
536     for l in self.layer3:
537         x = l(x)
538     #final relu
539     x = self.relu3(x)
540     #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
541     x = torch.flatten(x, 1)
542     #pass through our output layer
543     x = self.output(x)
544     return x
545
546 #want to set some hyperparameters
547 learning_rate_5 = 0.01
548 batch_size_5 = 128
549 num_epochs_5 = 50
550
551 #Make the CNN neural netowrk model
552 model_5 = Exercise5CNN().to(device)
553 #Our loss function will be cross entropy since we are getting a probability distribution
554 loss_5 = torch.nn.CrossEntropyLoss()
555 #Here we are going to use classic stochastic gradient descent without any special optimizations sinc
556 optimizer_5 = torch.optim.SGD(model_5.parameters(), lr= learning_rate_5)
557
558 #find the start time
559 start_5 = time.time()
560
561 #run the training loop
562 training_loss_5, training_accuracy_5, test_loss_5, test_accuracy_5 = training_loop(train_loader_cnn,
563 num_epochs_5, model_5, loss_5, optimizer_5)
564
565 #end time and get the total time
566 end_5 = time.time()
567 total_time = end_5 - start_5
568 #plotting
569 training_curve_plot("Deep CNN Model Performance 9 residual layers", training_loss_5, test_loss_5, tr
570 batch_size_5, learning_rate_5, total_time, num_epochs_5)
571
572 #5b deep neural network no residuals
573 #make the CNN residual network for exercise 5 according to the specifications in the assignment
574
575 #residual architecture for the 8 channels to 8 channels
576 class FirstKernelResidual5b(torch.nn.Module):
577     def __init__(self):

```



```

578         super().__init__()
579         #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
580         self.conv = torch.nn.Conv2d(in_channels = 8, out_channels = 8, kernel_size = 3, stride = 1, padding = 1)
581         #here are the relu and pooling layers
582         self.relu = torch.nn.ReLU()
583
584     def forward(self, x):
585         #modeling the books residual connection from section 11.2 figure 11.5b
586         # pass it through ReLU
587         x = self.relu(x)
588         #then we pass it through our convolutional layer where we relu then pool
589         x = self.conv(x)
590         return x
591
592 #residual architecture for the 16 channels to 16 channels
593 class SecondKernelResidual5b(torch.nn.Module):
594     def __init__(self):
595         super().__init__()
596         #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
597         self.conv = torch.nn.Conv2d(in_channels = 16, out_channels = 16, kernel_size = 3, stride = 1, padding = 1)
598         #here are the relu and pooling layers
599         self.relu = torch.nn.ReLU()
600
601     def forward(self, x):
602         #modeling the books residual connection from section 11.2 figure 11.5b
603         #pass through relu
604         temp = self.relu(x)
605         #then we pass it through our convolutional layer where we relu then pool
606         x = self.conv(x)
607         return x
608
609 #residual connections for 32 channels to 32 channels
610 class ThirdKernelResidual5b(torch.nn.Module):
611     def __init__(self):
612         super().__init__()
613         #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
614         self.conv = torch.nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size = 3, stride = 1, padding = 1)
615         #here are the relu and pooling layers
616         self.relu = torch.nn.ReLU()
617
618     def forward(self, x):
619         #modeling the books residual connection from section 11.2 figure 11.5b
620         #we pass it through ReLU
621         x = self.relu(x)
622         #then we pass it through our convolutional layer where we relu then pool
623         x = self.conv(x)
624         return x
625
626
627 class Exercise5bCNN(torch.nn.Module):
628     def __init__(self, first = 3, second = 3, third = 3):
629         super().__init__()
630         #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
631         self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1, padding = 1)
632         #adding many residual layers in this case default 3 more convolutions
633         self.layer1 = torch.nn.ModuleList([FirstKernelResidual5b() for _ in range(first)])
634         #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
635         self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)

```



```

636     #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
637     self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, padding= 1)
638     #adding many residual layers in this case default 3 more convolutions at the 16 channel length
639     self.layer2 = torch.nn.ModuleList([SecondKernelResidual5b() for _ in range(second)])
640     #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
641     self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
642     # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
643     self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, padding= 1)
644     #adding many residual layers in this case default 3 more convolutions at the 32 channel length
645     self.layer3 = torch.nn.ModuleList([ThirdKernelResidual5b() for _ in range(third)])
646     #non linearity
647     self.relu3 = torch.nn.ReLU()
648     #output network we have 32 channels and an image that is (7,7)
649     self.output = torch.nn.Linear(32 * 7 * 7, 10)
650
651     def forward(self, x):
652         #pass through the first convolution
653         #we don't need to relu because we pass it through several residual layers that will handle it
654         x = self.conv1(x)
655         #pass through several residual layers
656         for l in self.layer1:
657             x = l(x)
658         #pooling at the end of residual connections because pool will decrease our image size
659         x = self.pool1(x)
660         #pass through the second convolution to get to the next channel size of 16 and smaller image
661         x = self.conv2(x)
662         #pass through several residual layers
663         for l in self.layer2:
664             x = l(x)
665         #pooling to reduce size for the third convolutional layer
666         x = self.pool2(x)
667         #pass through the final convolution
668         x = self.conv3(x)
669         #pass through more residual convolutions then
670         for l in self.layer3:
671             x = l(x)
672         #final relu
673         x = self.relu3(x)
674         #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
675         x = torch.flatten(x, 1)
676         #pass through our output layer
677         x = self.output(x)
678         return x
679
680     #want to set some hyperparameters
681     learning_rate_5b = 0.01
682     batch_size_5b = 128
683     num_epochs_5b = 50
684
685     #Make the CNN neural network model
686     model_5b = Exercise5bCNN().to(device)
687     #Our loss function will be cross entropy since we are getting a probability distribution
688     loss_5b = torch.nn.CrossEntropyLoss()
689     #Here we are going to use classic stochastic gradient descent without any special optimizations since
690     optimizer_5b = torch.optim.SGD(model_5b.parameters(), lr= learning_rate_5b)
691
692     #find the start time
693     start_5b = time.time()

```

```

694
695 #run the training loop
696 training_loss_5b, training_accuracy_5b, test_loss_5b, test_accuracy_5b = training_loop(train_loader,
697 num_epochs_5b, model_5b, loss_5b, optimizer_5b)
698
699 #end time and get the total time
700 end_5b = time.time()
701 total_time = end_5b - start_5b
702 #plotting
703 training_curve_plot("Deep CNN Model Performance 9 extra layers no residuals", training_loss_5b, test,
704 batch_size_5b, learning_rate_5b, total_time, num_epochs_5b)
705
706 # Exercise 6a
707 # Adding batch normalization to residual layers
708 #make the CNN residual network for exercise 6 with batch normalization added which is a regularizati
709
710 #residual architecture for the 8 channels to 8 channels
711 class FirstKernelResidual6a(torch.nn.Module):
712     def __init__(self):
713         super().__init__()
714         #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
715         self.conv = torch.nn.Conv2d(in_channels = 8, out_channels = 8, kernel_size = 3, stride = 1,
716         #also want to batch normalize the input
717         #takes input (N,C,H,W) where C is channels I think so in this case 8
718         self.norm = torch.nn.BatchNorm2d(8)
719         #here are the relu and pooling layers
720         self.relu = torch.nn.ReLU()
721
722     def forward(self, x):
723         #modeling the books residual connection from section 11.2 figure 11.5b and 11.6 for batch no
724         #first we batch normalize
725         temp = self.norm(x)
726         #then we pass it through ReLU
727         temp = self.relu(temp)
728         #then we pass it through our convolutional layer where we relu then pool
729         temp = self.conv(temp)
730         #add in our residual connection
731         x = x + temp
732         return x
733
734 #residual architecture for the 16 channels to 16 channels
735 class SecondKernelResidual6a(torch.nn.Module):
736     def __init__(self):
737         super().__init__()
738         #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
739         self.conv = torch.nn.Conv2d(in_channels = 16, out_channels = 16, kernel_size = 3, stride = 1
740         #also want to batch normalize the input
741         #takes input (N,C,H,W) where C is channels I think so in this case 8
742         self.norm = torch.nn.BatchNorm2d(16)
743         #here are the relu and pooling layers
744         self.relu = torch.nn.ReLU()
745
746     def forward(self, x):
747         #modeling the books residual connection from section 11.2 figure 11.5b
748         #first we batch normalize
749         temp = self.norm(x)
750         #then we pass it through ReLU
751         temp = self.relu(temp)

```

```

752         #then we pass it through our convolutional layer where we relu then pool
753         temp = self.conv(temp)
754         #add in our residual connection
755         x = x + temp
756         return x
757
758     #residual connections for 32 channels to 32 channels
759     class ThirdKernelResidual6a(torch.nn.Module):
760         def __init__(self):
761             super().__init__()
762             #mimicks the first convolution but here we will just go from 8 inputs to 8 outputs
763             self.conv = torch.nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size = 3, stride = 1, padding = 1)
764             #also want to batch normalize the input
765             #takes input (N,C,H,W) where C is channels I think so in this case 8
766             self.norm = torch.nn.BatchNorm2d(32)
767             #here are the relu and pooling layers
768             self.relu = torch.nn.ReLU()
769
770         def forward(self, x):
771             #modeling the books residual connection from section 11.2 figure 11.5b
772             #first we batch normalize
773             temp = self.norm(x)
774             #then we pass it through ReLU
775             temp = self.relu(temp)
776             #then we pass it through our convolutional layer where we relu then pool
777             temp = self.conv(temp)
778             #add in our residual connection
779             x = x + temp
780             return x
781
782
783     class Exercise6aCNN(torch.nn.Module):
784         def __init__(self, first = 3, second = 3, third = 3):
785             super().__init__()
786             #1 input channel, 8 output channels, kernel size 3, stride 1, padding 1
787             self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 3, stride = 1, padding = 1)
788             #adding many residual layers in this case default 3 more convolutions
789             self.layer1 = torch.nn.ModuleList([FirstKernelResidual6a() for _ in range(first)])
790             #first pooling layer with kernel size 2, stride 2 reduces image to (8,14,14)
791             self.pool1 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
792             #8 input channels, 16 output channels, kernel size 3, stride 1 padding 1
793             self.conv2 = torch.nn.Conv2d(in_channels= 8,out_channels= 16 , kernel_size= 3, stride= 1, padding = 1)
794             #adding many residual layers in this case default 3 more convolutions at the 16 channel length
795             self.layer2 = torch.nn.ModuleList([SecondKernelResidual6a() for _ in range(second)])
796             #second pooling layer with kernel size 2, stride 2 reduces image to (16,7,7)
797             self.pool2 = torch.nn.MaxPool2d(kernel_size = 2, stride = 2)
798             # 16 inputs, 32 outputs, kernel size 3, stride 1, padding 1
799             self.conv3 = torch.nn.Conv2d(in_channels= 16,out_channels= 32 , kernel_size= 3, stride= 1, padding = 1)
800             #adding many residual layers in this case default 3 more convolutions at the 32 channel length
801             self.layer3 = torch.nn.ModuleList([ThirdKernelResidual6a() for _ in range(third)])
802             #non linearity
803             self.relu3 = torch.nn.ReLU()
804             #output network we have 32 channels and an image that is (7,7)
805             self.output = torch.nn.Linear(32 * 7 * 7, 10)
806
807         def forward(self, x):
808             #pass through the first convolution
809             #we don't need to relu because we pass it through several residual layers that will handel the non-linearity

```

```

810     x = self.conv1(x)
811     #pass through several residual layers
812     for l in self.layer1:
813         x = l(x)
814     #pooling at the end of residual connections because pool will decrease our image size
815     x = self.pool1(x)
816     #pass through the second convolution to get to the next channel size of 16 and smaller image
817     x = self.conv2(x)
818     #pass through several residual layers
819     for l in self.layer2:
820         x = l(x)
821     #pooling to reduce size for the third convolutional layer
822     x = self.pool2(x)
823     #pass through the final convolution
824     x = self.conv3(x)
825     #pass through more residual convolutions then
826     for l in self.layer3:
827         x = l(x)
828     #final relu
829     x = self.relu3(x)
830     #flatten all dimensions except batch dimension which is dimension 0 so we start at 1
831     x = torch.flatten(x, 1)
832     #pass through our output layer
833     x = self.output(x)
834     return x
835
836 #want to set some hyperparameters
837 learning_rate_6a = 0.025
838 batch_size_6a = 128
839 num_epochs_6a = 50
840
841 #Make the CNN neural netowrk model
842 model_6a = Exercise6aCNN().to(device)
843 #Our loss function will be cross entropy since we are getting a probability distribution
844 loss_6a = torch.nn.CrossEntropyLoss()
845 #Here we are going to use classic stochastic gradient descent without any special optimizations sinc
846 optimizer_6a = torch.optim.SGD(model_6a.parameters(), lr= learning_rate_6a)
847
848 #find the start time
849 start_6a = time.time()
850
851 #run the training loop
852 training_loss_6a, training_accuracy_6a, test_loss_6a, test_accuracy_6a = training_loop(train_loader,
853 num_epochs_6a, model_6a, loss_6a, optimizer_6a)
854
855 #end time and get the total time
856 end_6a = time.time()
857 total_time = end_6a - start_6a
858 #plotting
859 training_curve_plot("Deep CNN Model Performance Batch Norm", training_loss_6a, test_loss_6a, training_
860 batch_size_6a, learning_rate_6a, total_time, num_epochs_6a)
861
862 # Exercise 6b
863 # Modifying Adam parameters
864 #want to set some hyperparameters
865 learning_rate_6b = 0.0005
866 batch_size_6b = 128
867 num_epochs_6b = 35

```

868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917

```
#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, train
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, n
```
