SYDE 572 Assignment 4

Kritik Kaushal, k3kausha@uwaterloo.ca, 20854655

Exercise 1

1. The logistic regression algorithm was implemented to classify the first 2 classes of the MNIST dataset. Snippets of the code are shown below. In particular, the sigmoid function and gradient descent algorithm are shown below. Full code can be found in the Python file attached named "a4q1.py".

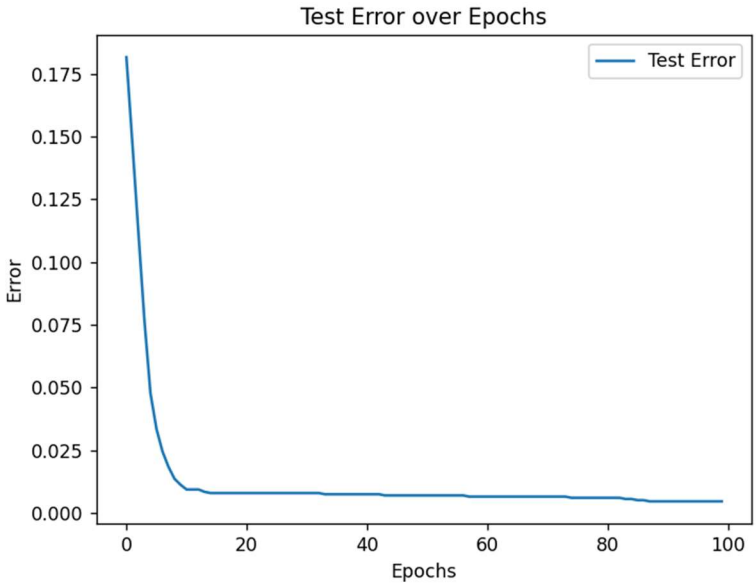
```python
def sigmoid(z):
  return 1 / (1 + np.exp(-z))
```

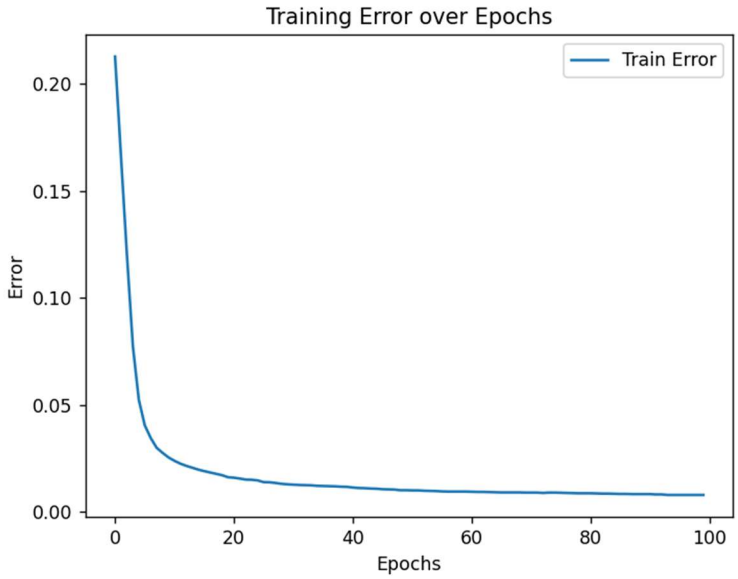
```python
for epoch in range(num_epochs):
  # calculate the "z" to feed into the sigmoid function for both train and test sets
  z = np.dot(training_data, w) + w0
  z_test = np.dot(test_data, w) + w0

  p_c1 = sigmoid(z)
  predictions_test = sigmoid(z_test)

  # calculate training losses and store in array
  loss = -np.mean(training_labels * np.log(p_c1) + (1 - training_labels) * np.log(1 - p_c1))
  losses.append(loss)

  # calculate test losses and store in array
  test_loss = -np.mean(test_labels * np.log(predictions_test) + (1 - test_labels) * np.log(1 - predictions_test))
  test_losses.append(test_loss)

  # calculate training accuracy and store error (1-accuracy) in array
  predictions = (p_c1 >= 0.5).astype(int)
  accuracy = np.mean(predictions == training_labels)
  accuracies.append(1-accuracy)

  # calculate test accuracy and store error (1-accuracy) in array
  predictions_test = (predictions_test >=0.5).astype(int)
  test_accuracy = np.mean(predictions_test == test_labels)
  test_accuracies.append(1-test_accuracy)

  # calculate gradients
  gradient_w = np.dot(training_data.T, (p_c1 - training_labels)) / len(training_labels)
  gradient_w0 = np.sum(p_c1 - training_labels) / len(training_labels)

  # update parameters based on gradients
  w -= learning_rate * gradient_w
  w0 -= learning_rate * gradient_w0
```

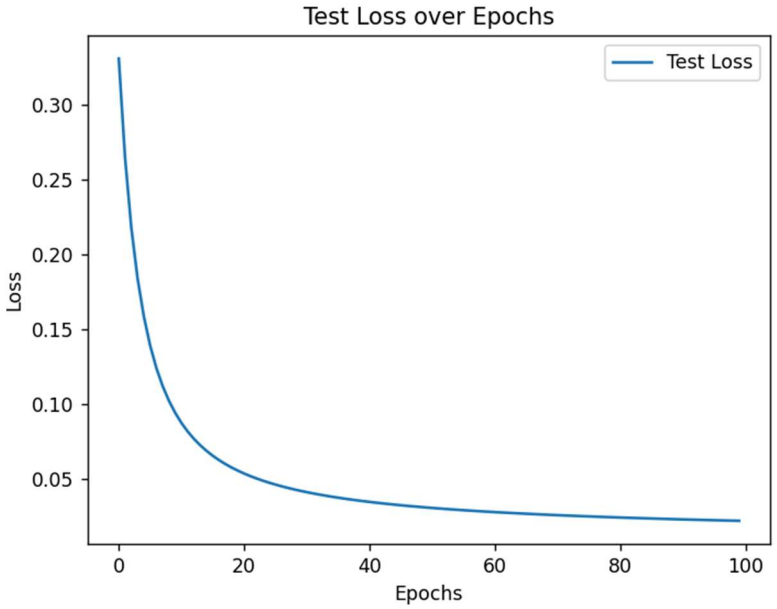
The following hyperparameters were used:

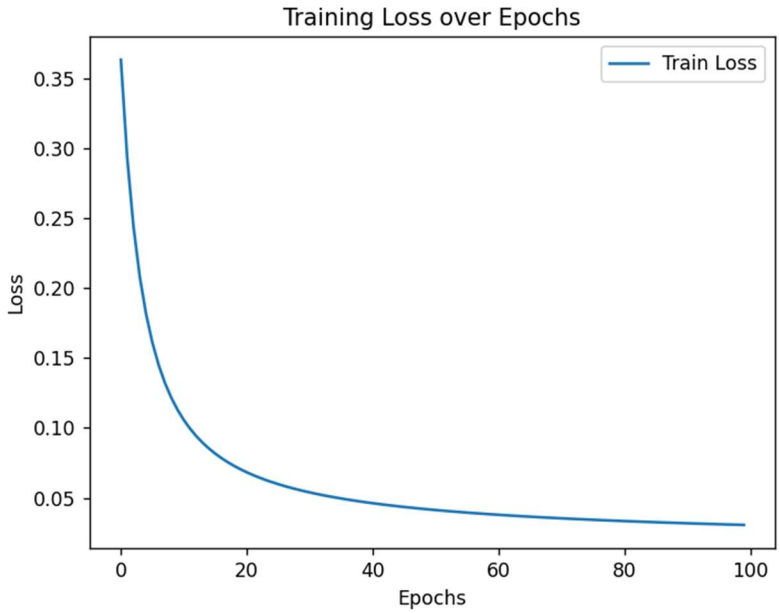
```python
learning_rate = 0.1
num_epochs = 100
```

2. After the training process with 100 epochs and a step size of 0.01, the following graphs are produced.

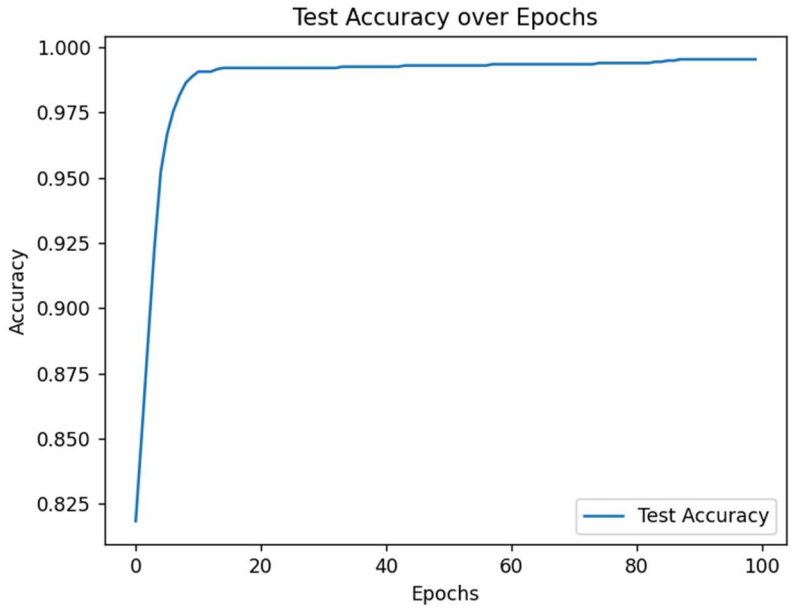


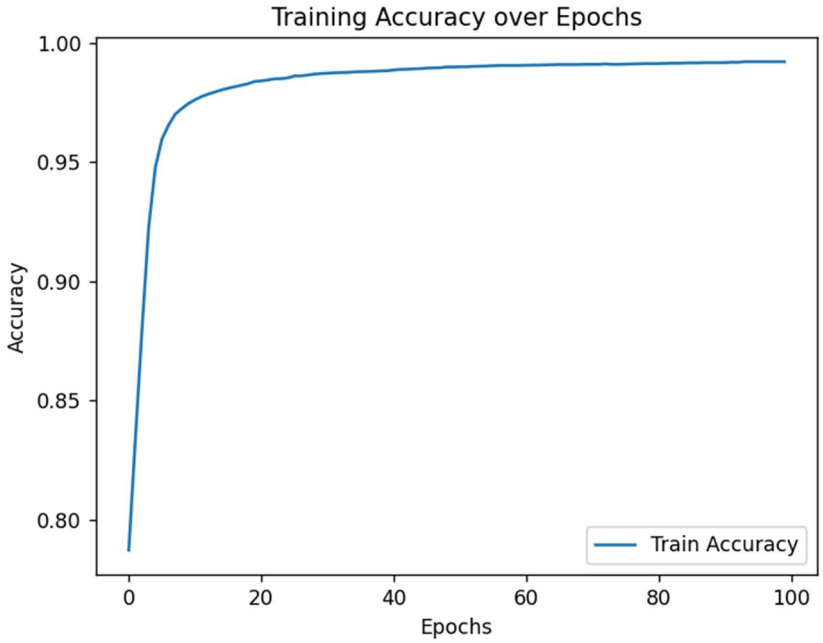






Additionally, for illustration purposes, the training and test accuracies were plotted over the number of epochs.





As the number of epochs increases, the train and test errors decrease (or phrased differently, the train and test accuracies approach 100%).

Also, the test and training losses are minimized as the number of epochs increases, which makes sense. The whole point of gradient descent is to minimize the loss function, and this is precisely the behaviour we observe.

3. Comparing the final test error to the other classifiers, the logistic regression performed the best. This makes sense since logistic regression models are very well suited to binary classification problems.

   Logistic regression does not rely on any distance metrics and instead uses a logistic function (sigmoid) to determine probability of belonging to a class. Logistic regression is less sensitive to feature scaling because it uses a logistic function. For these reasons, the logistic regression model has a much higher accuracy than the other methods.

Exercise 2

1. The VGG11 network architecture was implemented. The full code can be found in "a4q2.py". The original MNIST images were reshaped from 28x28 to 32x32. This is because for the VGG11 architecture, the minimum input size is 32x32 as defined in [1]. This is due to the eventual max-pooling layers, which reduce the number of dimensions. Any image smaller than 32x32 will lose too much data from the pooling layers, which is why we need to increase the size of the input images from 28x28 to 32x32.

   The implementation of the VGG11 architecture can be found in "a4q2.py".

   I am using the following version of Python and other relevant packages:

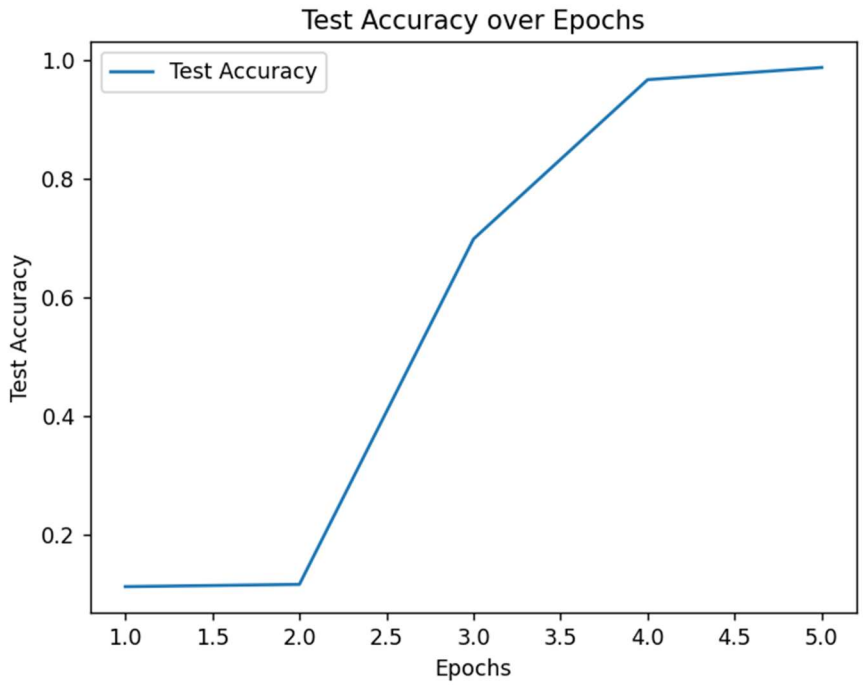
```
Python Version:  3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)]
Pytorch Version:  2.1.0+cu121
Matplotlib Version:  3.8.0
```

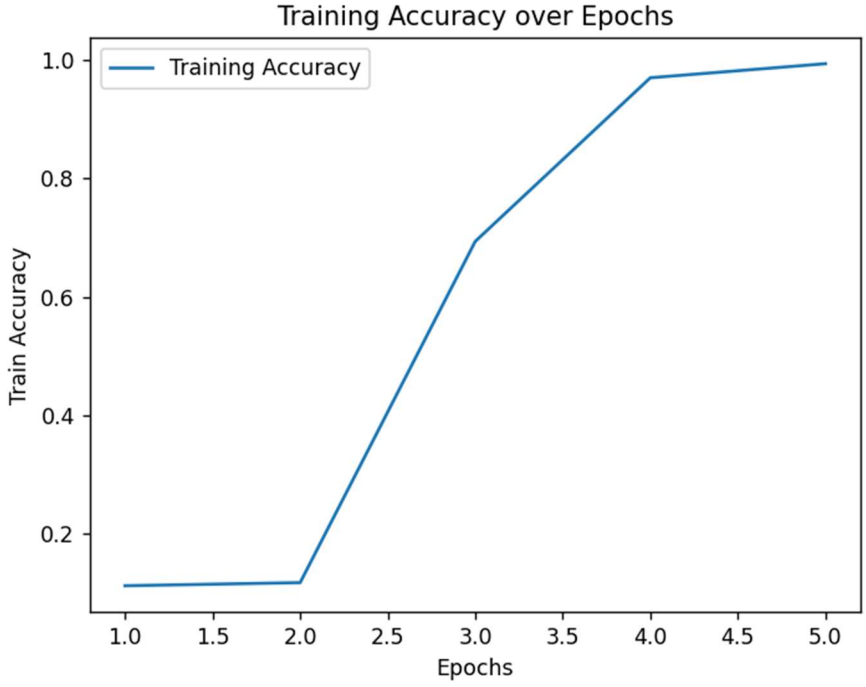
2. The plots of the test and training accuracies and test and training losses against the number of epochs are shown below. The CNN was run for 5 epochs to see the general trend.

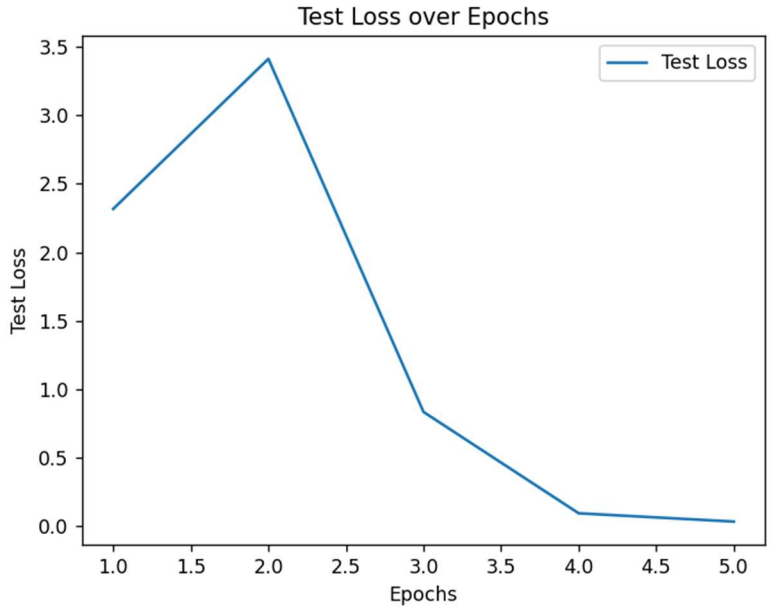
   I have a GPU in my laptop, so I used CUDA to run the net on my GPU. This greatly sped up my training and test times. Note that if you try running the scripts without GPU, 5 epochs may not be enough for the network to converge (due to floating point operation accuracy). I am running my code on the following GPU:

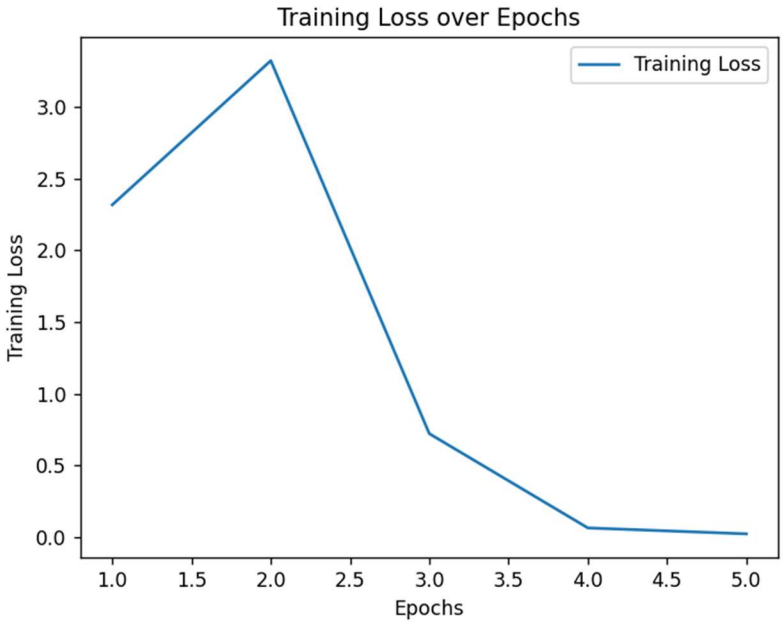
```
CUDA is available!
CUDA Device Name: NVIDIA GeForce GTX 1650
```

   The plots are shown below.


Test Accuracy over Epochs
Test Accuracy
1.0
0.8
0.6
0.4
0.2
Test Accuracy
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Epochs


Training Accuracy over Epochs
Training Accuracy
1.0
0.8
0.6
0.4
0.2
Train Accuracy
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Epochs

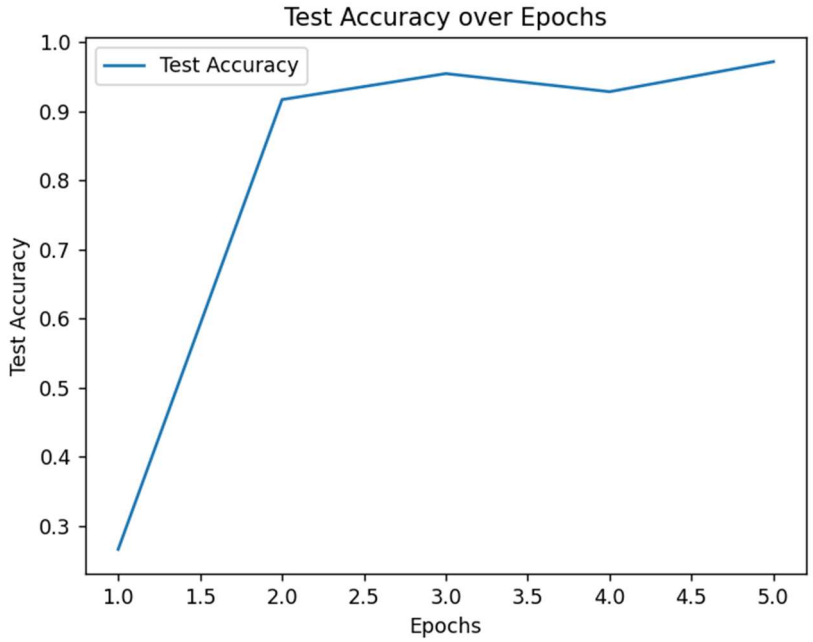


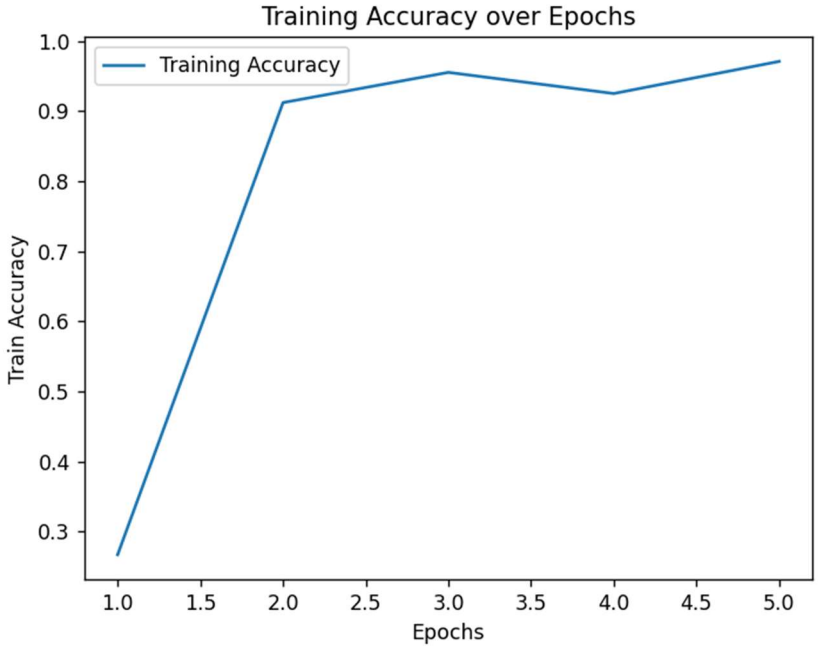


These plots make sense. As the number of epochs increases, the accuracies increase and the losses decrease, which is the expected behaviour. The numerical values of the training accuracy using SGD vs. number of epochs are shown below:

```
Epoch: 0, Training Accuracy: 0.11236666666666667
Epoch: 1, Training Accuracy: 0.11761666666666666
Epoch: 2, Training Accuracy: 0.6935833333333333
Epoch: 3, Training Accuracy: 0.96975
Epoch: 4, Training Accuracy: 0.9934666666666667
```

3. The Adam optimizer was used instead of Stochastic Gradient Descent. The graphs of training and test accuracies against the number of epochs were plotted, shown below.





Again, the CNN was run for 5 epochs. The numerical values of the training accuracy using Adam vs. number of epochs are shown below:

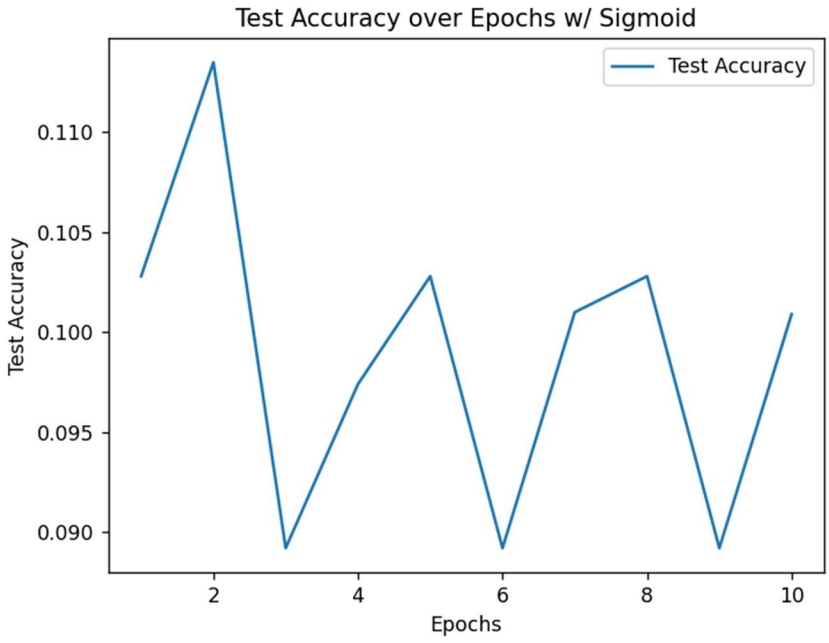
```
Epoch: 0, Training Accuracy: 0.26675
Epoch: 1, Training Accuracy: 0.9121833333333333
Epoch: 2, Training Accuracy: 0.9553166666666667
Epoch: 3, Training Accuracy: 0.92515
Epoch: 4, Training Accuracy: 0.971
```

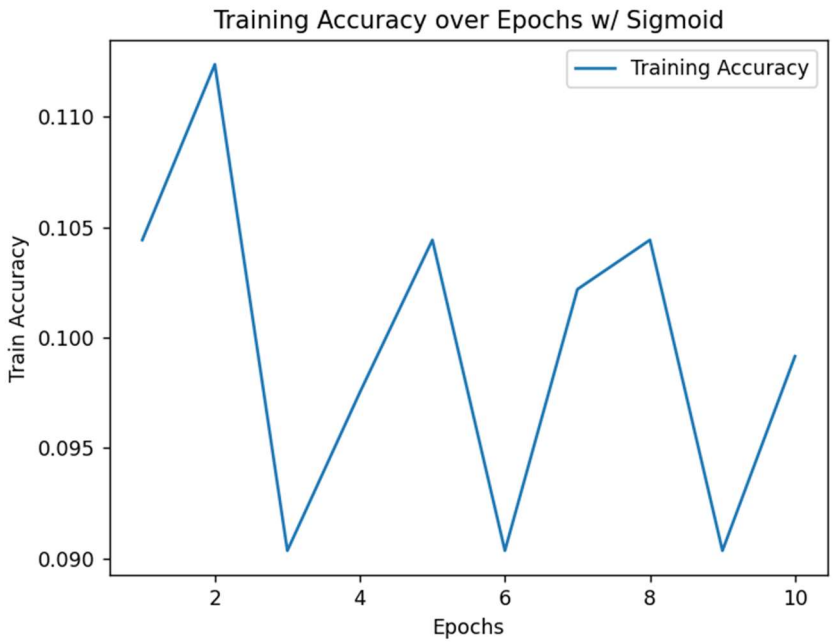
Compared to the training accuracies of the SGD optimizer, the Adam optimizer converges faster. By the second epoch, the accuracy was already 91%, whereas for the SGD optimizer, the accuracy at the second epoch is 12%.

However, the SGD optimizer achieves an accuracy of 99% by the 5th epoch, whereas the Adam optimizer achieves an accuracy of 97% by the 5th epoch.

Thus, the SGD optimizer performs better but the Adam optimizer converges faster. So, the choice of Adam or SGD depends on the constraints of the problem and if converging fast is preferred over the final accuracy. SGD performs better because there are fewer hyperparameters to tune, so it is simpler to achieve high accuracy. Adam does not reach as high accuracy as SGD because there are more hyperparameters to tune, but it can converge faster.

4. In the modified CNN, ReLU was replaced by sigmoid as the activation function. The CNN was trained for 10 epochs to get a better understanding of the behavior. The training and test accuracies were plotted as shown below.



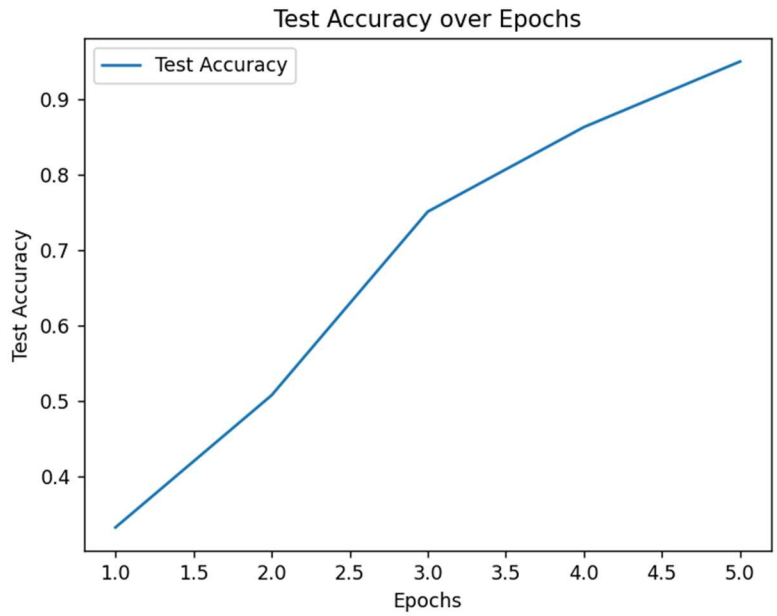


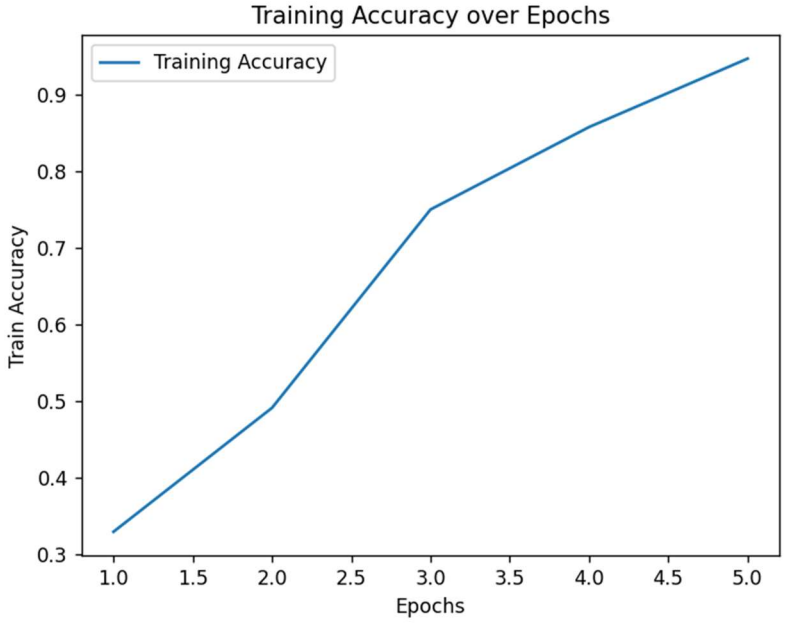
As can be seen, accuracy never converges, and never gets better. This makes sense, as using the sigmoid function as the activation function causes the vanishing gradients problem.

Since the sigmoid function "squeezes" the input between 0 and 1, a large change in the input only causes a small change in the output. This becomes an issue when the gradients are calculated during backpropagation.

If there are *n* layers, and they all have small derivatives due to the sigmoid function, then *n* small derivative values get multiplied together. This results in an even smaller number, and the gradient decreases rapidly as the backpropagation progresses. This small gradient can be an issue, as the weights and biases of the first layers will not be updated properly after each epoch. This makes it hard for the network to learn anything about the data, resulting in poor accuracy and never converging (as we see in the graphs above).

5. The Dropout was removed from the CNN. The CNN was trained for 5 epochs, and the training and test accuracies were plotted as shown below.





By removing the Dropout, we see some interesting behaviour. Dropout is a regularization technique which encourages all units to share the load to make the correct prediction by perturbing the network. In the graph above, we see that compared to the first graphs, the network is slower to reach high accuracy. It takes longer for the network to converge.

This makes sense, since dropout makes the network more efficient and robust to noisy data, which may help it converge faster. In the implementation with dropout, we see exactly that. The network seems to be more robust, leading to better accuracy much faster than the case where dropout was not applied.

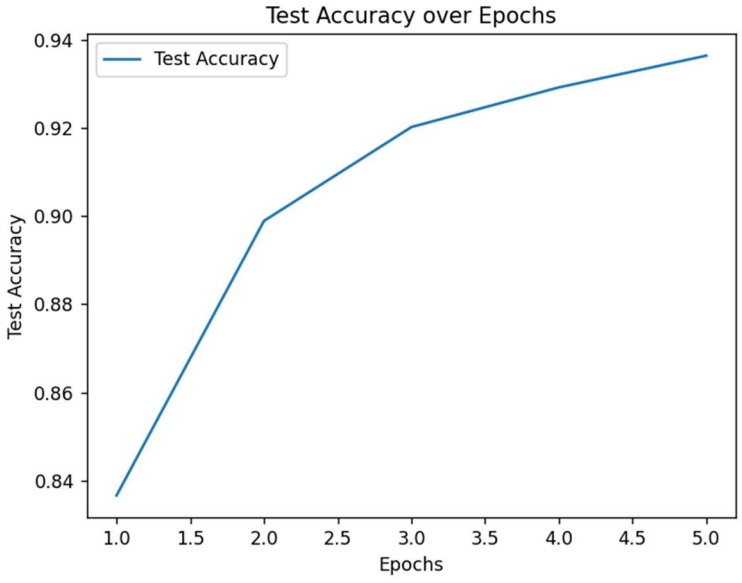
Exercise 3

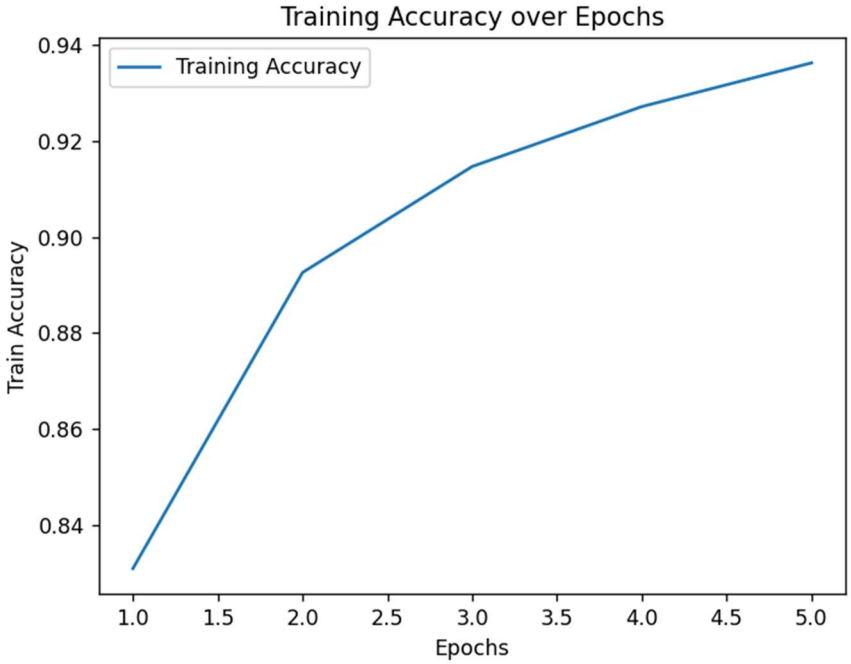
1. A 3-layer MLP was implemented. The full code can be found in the file called “a4q3.py”. A snippet of the code showing just the MLP architecture is shown below:

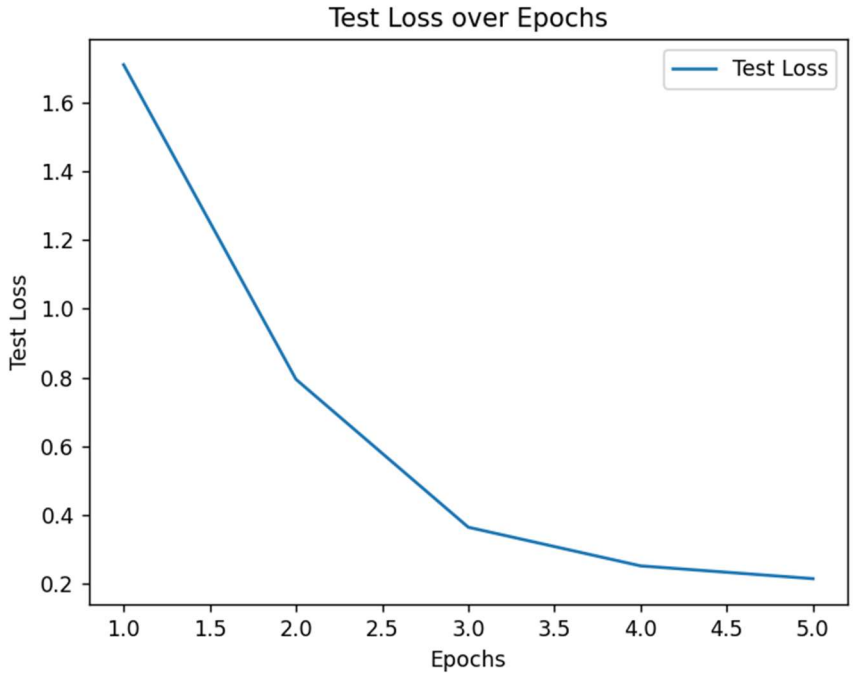
```python
# 3 Layer MLP implementation
class MLP(nn.Module):
  def __init__(self):
    super(MLP, self).__init__()
    # MLP layers
    self.layers = nn.Sequential(
      nn.Linear(784, 512),
      nn.ReLU(),
      nn.Linear(512, 512),
      nn.BatchNorm1d(512),
      nn.ReLU(),
      nn.Linear(512, 10)
    )

  def forward(self, x):
    # flatten the input to (28x28, 1) = (784, 1)
    x = torch.flatten(x, start_dim=1)
    x = self.layers(x)
    return x
```

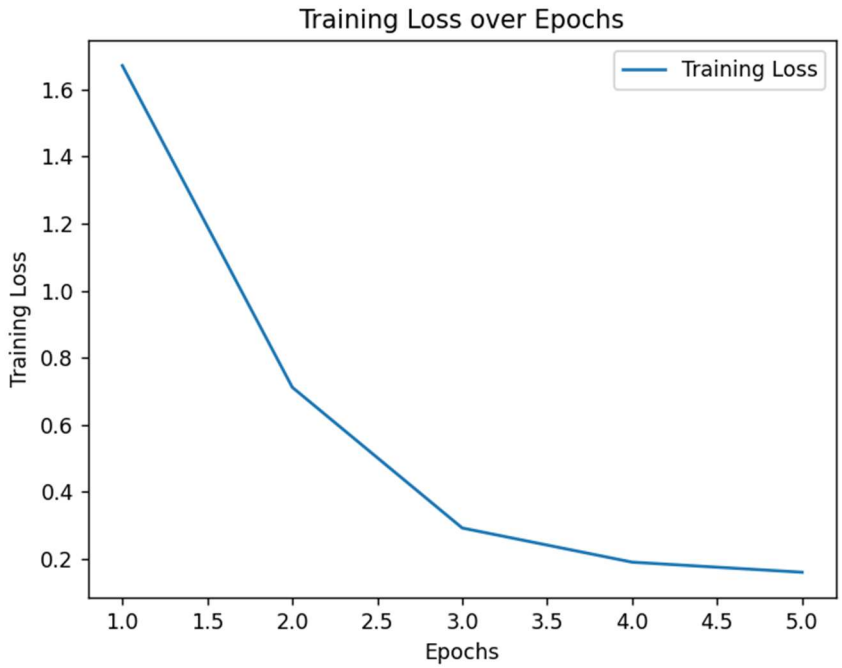
   The input images are flattened to 784x1, which is the number of input channels in the first layer of the MLP.

2. The plots of the test and training accuracies and test and training losses against the number of epochs are shown below. The MLP was run for 5 epochs to see the general trend.




Training Accuracy over Epochs
Training Accuracy
Train Accuracy
Epochs
0.94
0.92
0.90
0.88
0.86
0.84
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


Test Loss over Epochs
Test Loss
Test Loss
Epochs
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0



Also, the numerical values of the training accuracy against the number of epochs are provided below:

```
Epoch: 0, Training Accuracy: 0.83085
Epoch: 1, Training Accuracy: 0.8926166666666666
Epoch: 2, Training Accuracy: 0.9147333333333333
Epoch: 3, Training Accuracy: 0.9272333333333334
Epoch: 4, Training Accuracy: 0.93635
```

3. Compared to the VGG11, the MLP has a higher starting accuracy than VGG11. For the first epoch, the MLP had an accuracy of 83% whereas the VGG11 had an accuracy of 11%.

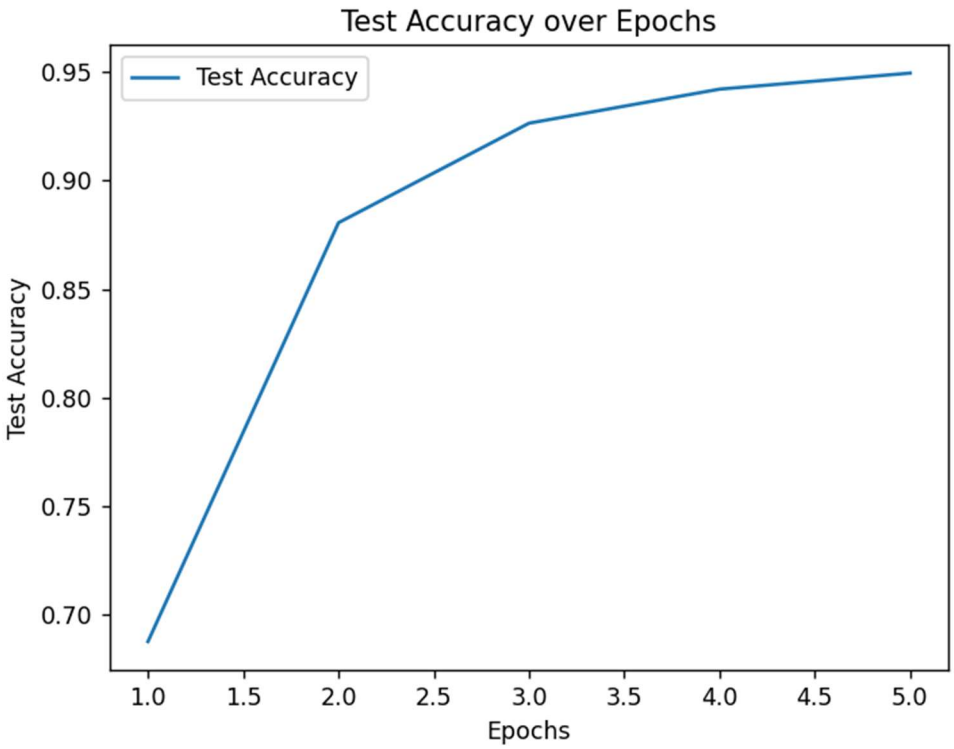
   However, the VGG11 net has a much higher accuracy after the 5^{th} epoch. After the 5^{th} epoch, the accuracy of the MLP is 94%, whereas for the VGG11 it is 99%.

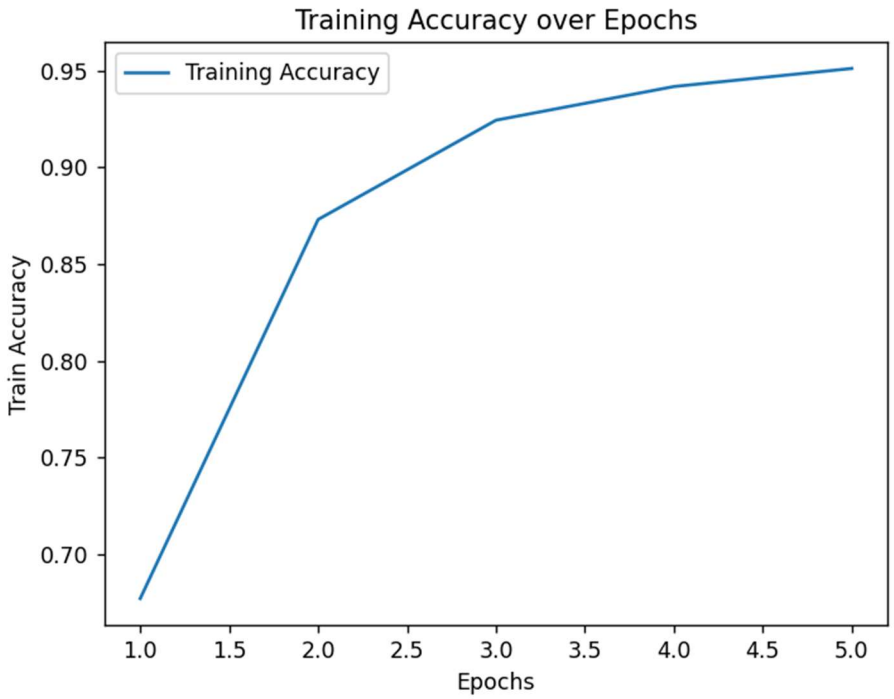
   For the MLP, the loss vs the number of epochs seems to be approaching some non-zero number asymptotically. For the VGG11 net, the loss vs number of epochs approaches zero. Based on this information, the VGG11 architecture seems to be better for MNIST classification. It has a higher accuracy after 5 epochs, and the loss function seems to be truly minimized.

4. Another layer was added to the MLP. The modified architecture is as follows:

```python
# 3 Layer MLP implementation
class MLP(nn.Module):
  def __init__(self):
    super(MLP, self).__init__()
    # MLP layers
    self.layers = nn.Sequential(
      nn.Linear(784, 512),
      nn.ReLU(),
      nn.Linear(512, 512),
      nn.BatchNorm1d(512),
      nn.ReLU(),
      nn.Linear(512, 512),
      nn.BatchNorm1d(512),
      nn.ReLU(),
      nn.Linear(512, 10)
    )

  def forward(self, x):
    # flatten the input to (28x28, 1) = (784, 1)
    x = torch.flatten(x, start_dim=1)
    x = self.layers(x)
    return x
```

The test and training accuracies were plotted against the number of epochs, and the results are as follows:





The numerical values of the training accuracy are also provided:

```
Epoch: 0, Training Accuracy: 0.6772333333333334
Epoch: 1, Training Accuracy: 0.8731333333333333
Epoch: 2, Training Accuracy: 0.92445
Epoch: 3, Training Accuracy: 0.9418333333333333
Epoch: 4, Training Accuracy: 0.9511833333333334
```

Compared to the previous MLP implementation, this modified version starts off at a much lower accuracy (67%, compared to 83% prior). However, after the 5th epoch, the modified MLP has a higher accuracy of 95%, compared to 94% prior.

This is interesting, as the modified MLP seems to be improving and learning at a better rate than the previous version of MLP. The modified MLP performs better because adding an additional layer with ReLU introduces more non-linearities in the network. This allows the network to learn more complex non-linear relations, which could help improve accuracy. In our case, we've added another layer with ReLU, so we have increase the net's ability to learn non-linear relations.

References

[1] PyTorch, "VGG11," PyTorch, 2017. [Online]. Available: https://pytorch.org/vision/main/models/generated/torchvision.models.vgg11.html.