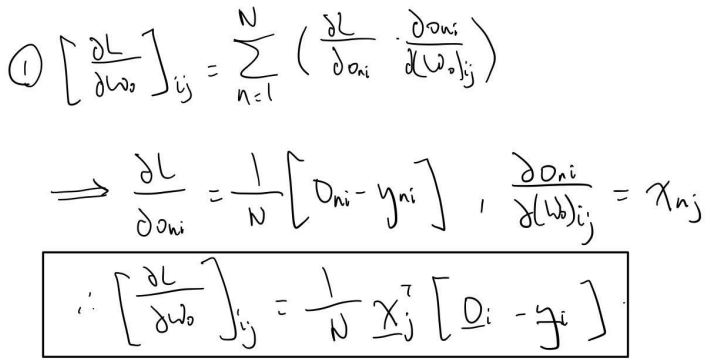
1 Neural Networks using Numpy

1.1 Helper Functions

```
def relu(x):
  return np.maximum(x, 0)

def softmax(x):
  x -= np.max(x, axis=-1, keepdims=True) # axis should be last
  return np.exp(x) / np.sum( np.exp(x), axis=-1, keepdims=True) # axis
should be last

def computeLayer(X, W, b):
  return X @ W + b

def CE(target, prediction):
  return -np.mean(target*np.log(prediction))

def gradCE(target, prediction):
  #prediction = softmax(o)
  return prediction - target #di(L)/di(o)
```

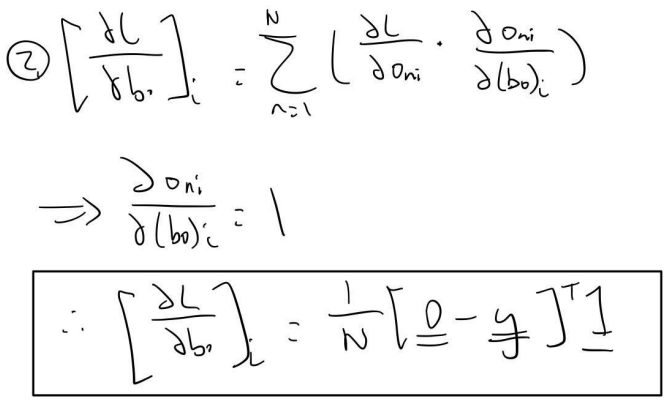
1.2 Backpropagation Derivation

1.2.1 Gradient of loss with respect to outer layer weights



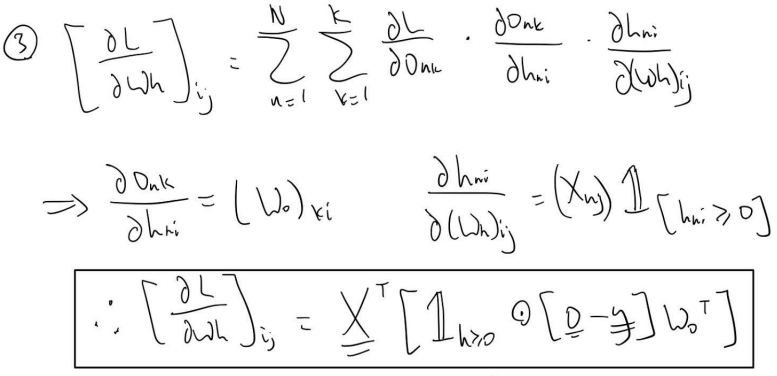
```
def gradOuterWeights(target, prediction, hiddenPrediction):
  #di(L)/di(o) * di(o)/di(Wo) = di(L)/di(Wo)
  return hiddenPrediction.T @ gradCE(target, prediction)
```

1.2.2 Gradient of loss with respect to outer layer biases



```
def gradOuterBias(target, prediction):
  return np.sum(gradCE(target, prediction), axis = 0)
```

1.2.3 Gradient of loss with respect to hidden layer weights



```
def gradInnerWeights (target, prediction, x, Wo, hiddenPrediction):
  #hiddenPrediction = Whx + bh
  reluGrad = (hiddenPrediction > 0).astype(np.int32) #gradient of ReLU
  return x.T @ ( reluGrad * ((gradCE(target, prediction) @ Wo.T)) )
```

1.2.4 Gradient of loss with respect ot hidden layer biases

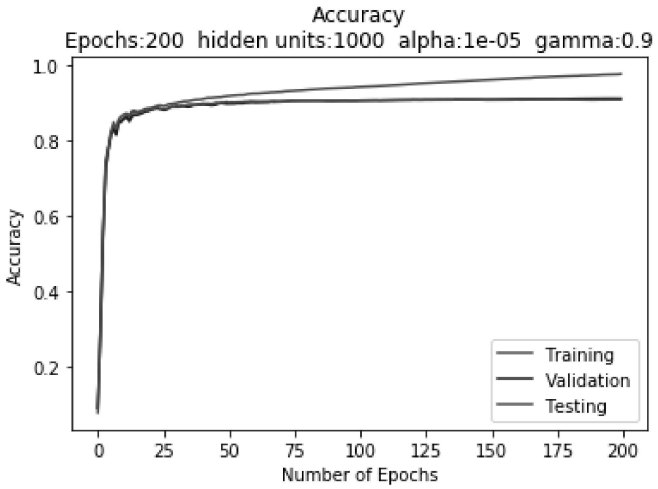
$$\textcircled{4}\quad \left[\frac{\partial L}{\partial b_h}\right]_i = \sum_{n=1}^{N}\sum_{k=1}^{K} \frac{\partial L}{\partial o_{nk}} \cdot \frac{\partial o_{nk}}{\partial h_{ni}} \cdot \frac{\partial h_{ni}}{\partial b_h}$$

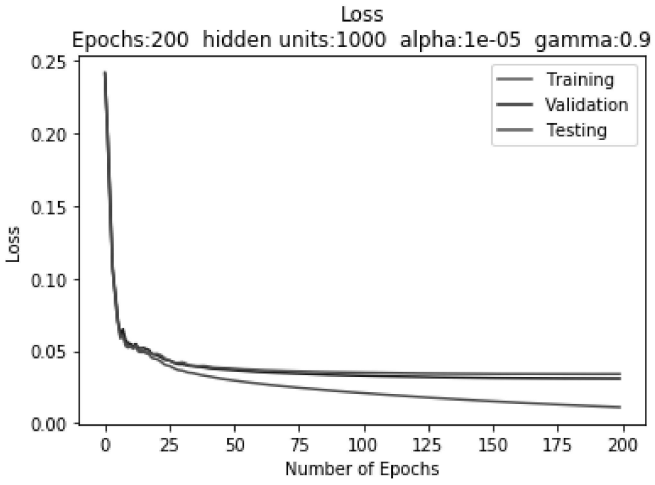
$$\Rightarrow \frac{\partial h_{ni}}{\partial b_h} = \mathbb{1}_{[h_{ni}>0]}$$

$$\therefore \left[\frac{\partial L}{\partial b_h}\right]_i = \left[\mathbb{1}_{h>0} \odot \left[\underline{0} - \underline{\hat{y}}\right] W_o^T\right] \cdot \underline{1}$$

```
def gradInnerBias(target, hiddenPrediction, prediction, Wo):
  reluGrad = (hiddenPrediction > 0).astype(np.int32) #gradient of ReLU
  return np.sum( reluGrad * ((gradCE(target, prediction) @ Wo.T)), axis=0)
```

1.3 Learning





	Training	Test	Valid
Final Accuracy	0.9778	0.9097	0.9125
Final Loss	0.0110	0.0340	0.0308

Neural network training code

```
def TrainNN(trainData, testData, validData, trainTarget, testTarget, validTarget, epochs,
hidden_units, alpha, gamma):
  # Reshape data
  trainData = trainData.reshape( trainData.shape[0], trainData.shape[1]**2 )
  testData = testData.reshape( testData.shape[0], testData.shape[1]**2 )
  validData = validData.reshape( validData.shape[0], validData.shape[1]**2 )
```

```
  # Initialize weights, bias and v for hidden and output layers
  h_weight, h_bias, h_v_w, h_v_b = initWeightBiasV(0, trainData.shape[1], hidden_units)
  o_weight, o_bias, o_v_w, o_v_b = initWeightBiasV(0, hidden_units, 10)


  # Accuracy and loss accounting
  train_accuracy = []
  train_loss = []

  test_accuracy = []
  test_loss = []

  valid_accuracy = []
  valid loss = []


  for epoch in range(epochs):
    print("Epoch:", epoch)

    # Get forward prop prediction
    hiddenPrediction, trainPrediction = getForwardProp( trainData, h_weight, h_bias, o_weight,
o_bias )
    testHiddenPrediction, testPrediction = getForwardProp( testData, h_weight, h_bias,
o_weight, o_bias )
    validHiddenPrediction, validPrediction = getForwardProp( validData, h_weight, h_bias,
o_weight, o_bias )


    # Get accuracy and loss
    train_accuracy.append( getAccuracy(trainPrediction, trainTarget) )
    train_loss.append( CE(trainTarget, trainPrediction) )

    test_accuracy.append( getAccuracy(testPrediction, testTarget) )
    test_loss.append( CE(testTarget, testPrediction) )

    valid_accuracy.append( getAccuracy(validPrediction, validTarget) )
    valid_loss.append( CE(validTarget, validPrediction) )


    # Update weights
    h_weight, h_bias, h_v_w, h_v_b = updateHiddenWeights( alpha, gamma, trainData,
trainTarget, trainPrediction, hiddenPrediction, o_weight, h_bias, h_v_w, h_v_b, h_weight )
    o_weight, o_bias, o_v_w, o_v_b = updateOuterWeights( alpha, gamma, trainTarget,
trainPrediction, hiddenPrediction, o_weight, o_bias, o_v_w, o_v_b )

  # report accuracies and losses
  print("Final Train Data Accuracy:", train_accuracy[epochs-1])
  print("Final Train Data Loss:", train_loss[epochs-1])

  print("Final Test Data Accuracy:", test_accuracy[epochs-1])
  print("Final Test Data Loss:", test loss[epochs-1])

  print("Final Valid Data Accuracy:", valid accuracy[epochs-1])
  print("Final Valid Data Loss:", valid_loss[epochs-1])

  PlotAccuracyLoss(train_loss, test_loss, valid_loss, train_accuracy, test_accuracy,
valid_accuracy, epochs, hidden_units, alpha, gamma)
```

Helper functions for TrainNN

```
def initWeightBiasV(mean, in size, out size):
  ## Xavier Init weights for hidden and output layer
  # | w(0,0)          ...          w(0, output layer size -1) |
  # | w(input size, 0) ..  w(input size, output layer size -1)|
  stdev = np.sqrt( 2.0 / (in size + out size) )
  w = np.random.normal( mean, stdev, (in_size, out_size) )
```

```
  b = np.zeros( (out_size) ) # zeros is better
  v_w = np.full( w.shape, 1e-5 ) # too big by x10
  v_b = np.full( b.shape, 1e-5 ) # ditto
  return w, b, v_w, v_b

def getForwardProp(input, h_weight, h_bias, o_weight, o_bias):
  hiddenPrediction = relu( computeLayer(input, h_weight, h_bias) )
  NN_Output = softmax( computeLayer(hiddenPrediction, o_weight, o_bias) )
  return hiddenPrediction, NN_Output

def getAccuracy(prediction, target):
  # Get number of elements that are equal and return equal el / total el
  comparison = np.equal( np.argmax(prediction, axis=1), np.argmax(target,
axis=1) )
  return np.mean( comparison )

def updateHiddenWeights( alpha, gamma, input, target, prediction,
hiddenPrediction, o_weight, h_bias, h_v_w, h_v_b, h_weight ):
  # update weights
  grad_w = gradInnerWeights(target, prediction, input, o_weight,
hiddenPrediction)
  v_w_new = gamma * h_v_w + alpha * grad_w
  h_weight -= v_w_new

  # update bias
  grad_b = gradInnerBias(target, hiddenPrediction, prediction, o_weight)
  v_b_new = gamma * h_v_b + alpha * grad_b
  h_bias -= v_b_new

  return h_weight, h_bias, v_w_new, v_b_new

def updateOuterWeights(alpha, gamma, target, prediction, hiddenPrediction,
o_weight, o_bias, o_v_w, o_v_b ):
  # Update weights
  grad_w = gradOuterWeights(target, prediction, hiddenPrediction)
  v_w_new = gamma * o_v_w + alpha * grad_w
  o_weight -= v_w_new

  # Update bias
  grad_b = gradOuterBias(target, prediction)
  v_b_new = gamma * o_v_b + alpha * grad_b
  o_bias -= v_b_new

  return o_weight, o_bias, v_w_new, v_b_new
```

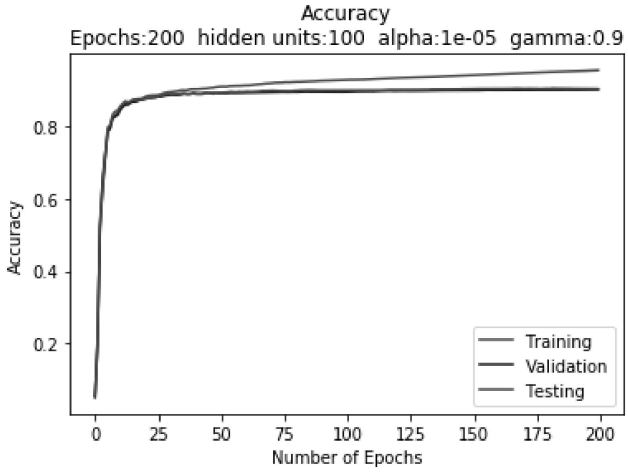
Utility functions for Train NN

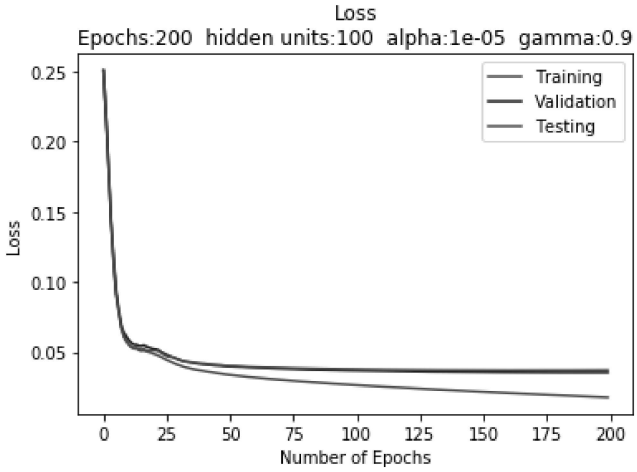
```
def PlotAccuracyLoss(train_loss, testing_loss, valid_loss, train_acc,
test_acc, validation_acc, epochs, hidden_units, alpha, gamma):
    x_axis = np.arange(0, epochs).tolist()

    #plot accuracies
    title = "Accuracy\nEpochs:" + str(epochs) + "  hidden units:" +
str(hidden_units) + "  alpha:" + str(alpha) + "  gamma:" + str(gamma)
    plt.title(title)
    plt.plot(x_axis, train_acc, 'r', label = "Training")
```

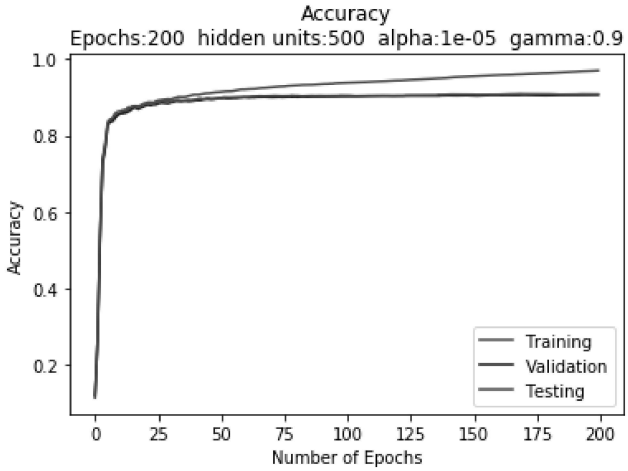
```
    plt.plot(x axis,validation acc,'b', label = "Validation")
    plt.plot(x axis,test acc,'g', label = "Testing")
    plt.legend(loc="lower right")
    plt.ylabel('Accuracy')
    plt.xlabel('Number of Epochs')
    plt.show()

    #plot losses
    title = "Loss\nEpochs:" + str(epochs) + "  hidden units:" +
str(hidden units) + "  alpha:" + str(alpha) + "  gamma:" + str(gamma)
    plt.title(title)
    plt.plot(x axis,train loss,'r', label = "Training")
    plt.plot(x_axis,valid_loss,'b', label = "Validation")
    plt.plot(x_axis,testing_loss,'g', label = "Testing")
    plt.legend(loc="upper right")
    plt.ylabel('Loss')
    plt.xlabel('Number of Epochs')
    plt.show()
```

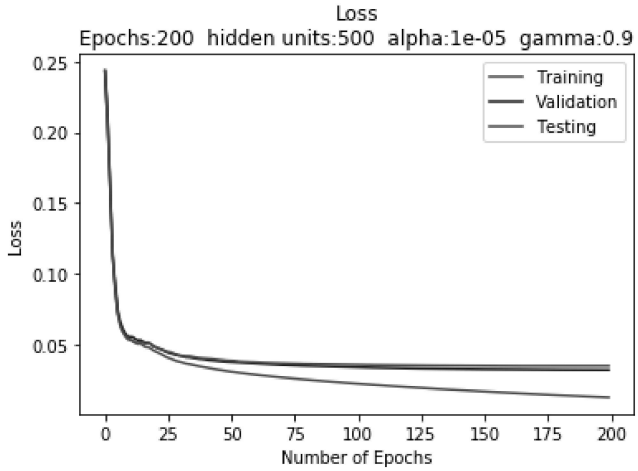
1.4 Hyperparameter Investigation

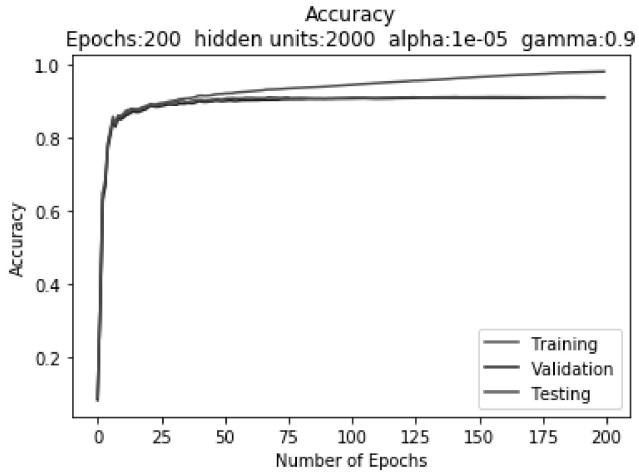
1.4.1 Number of Hidden Units

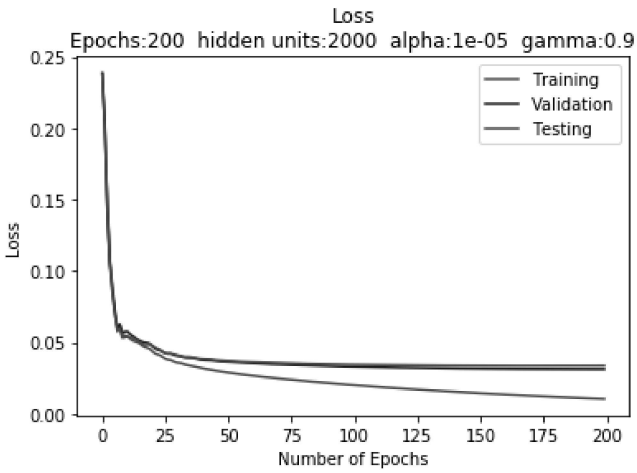












100 Hidden Units

	Training	Test	Valid
Final Accuracy	0.9536	0.9020	0.8997
Final Loss	0.0178	0.0373	0.0355

500 Hidden Units

	Training	Test	Valid
Final Accuracy	0.9705	0.9086	0.9067
Final Loss	0.0128	0.0351	0.0323

2000 Hidden Units

	Training	Test	Valid
Final Accuracy	0.9818	0.9115	0.9105
Final Loss	0.0103	0.0336	0.0312

As the number of hidden units increase, we notice a rise in the final training accuracy. At 100 hidden units, the training accuracy is 95.36%, and this number continues to rise when we use 500, 1000 (from section 1.3) and 2000 hidden units, ending with an accuracy of 98.18%. Overall, there is a 2.75% improvement in training accuracy when the number of hidden units increase from 500 through to 2000.

Examining the test and valid accuracies, we notice that there is also an upward trend in accuracies when increasing the number of hidden units used. However, the range of improvement is slightly lower: valid accuracy increased by 1.08%, whereas test accuracy only improved by 0.95%.

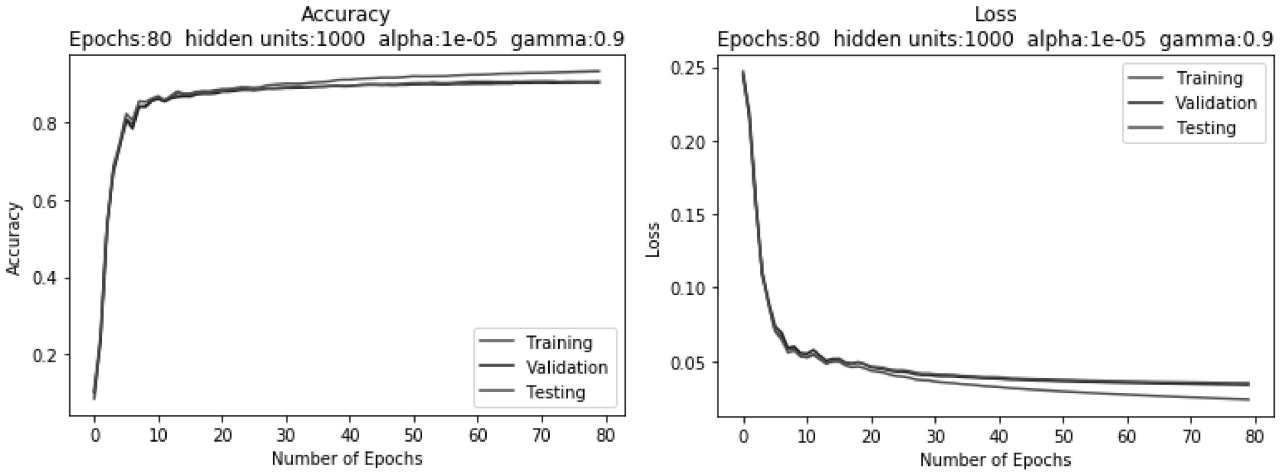
Similarly, it is observed that as the number of hidden units increase, training, test, and valid loss decrease. However, it can be observed that regardless of the number of hidden units used, loss in the testing data and valid data is around twice as large as the loss in training data.

To summarize, we have noticed that as hidden units increase, our accuracies and losses generally become better. However, we notice the most drastic improvements in training accuracy and loss, suggesting that our model may be overfitting to the training data.

1.4.2 Early Stopping

From the graphs presented in 1.4.1, we noticed that there is a general trend where the accuracy and loss of testing and valid data settle, and do not continue to improve as the training accuracy and loss do. Therefore, if we stop training when there is little improvement of accuracy and loss in valid and testing data, we should be able to achieve similar results in less run time.

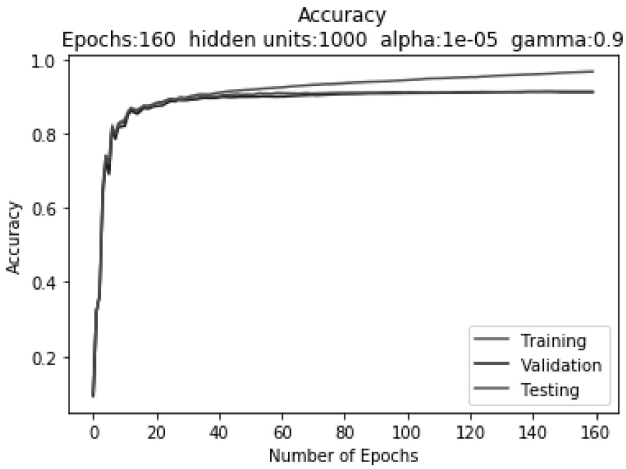
By observing the graphs in 1.3 and 1.4.1, we see that the valid and testing accuracies and losses settle after around 80 epochs. We will choose to stop at 80, and at 160.

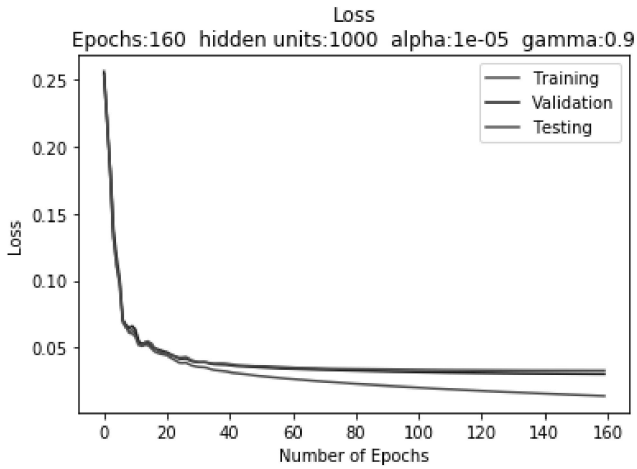


80 Epochs

	Training	Test	Valid
Final Accuracy	0.9342	0.9075	0.9050
Difference*	-4.56%	-0.24%	-0.82%

*Difference compared to values in 1.3





160 Epochs

	Training	Test	Valid
Final Accuracy	0.9671	0.9108	0.9132
Difference*	-1.09%	+0.12%	+0.08%

*Difference compared to values in 1.3

Looking at the resulting accuracies at 80 and 160 epochs, we see that the valid and test accuracies are within 1% of the final accuracy at 200 epochs. The more “drastic” improvements come from training accuracies. However, this is acceptable as we are trying to avoid overfitting to the training data.

2 Neural Networks in Tensorflow

For this section of the assignment, we have used Keras to implement the Neural Network.

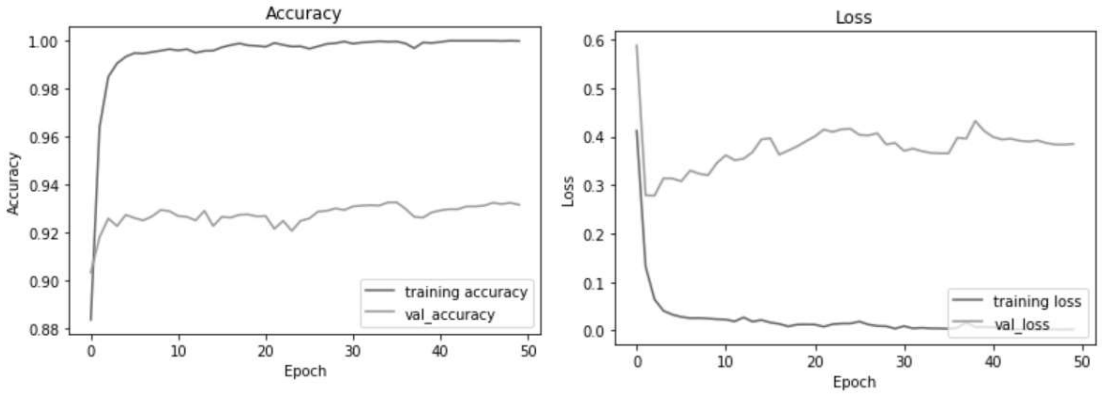
2.1 Model Implementation

```
from tensorflow.keras import datasets, layers, models, optimizers

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), padding="same",strides=1,
activation='relu', use_bias= True, bias_initializer='zeros',
kernel_initializer=initializers.glorot_normal(seed=14),
input_shape=(28, 28, 1)))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(784, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

```
model.compile(optimizer=optimizers.Adam(learning_rate=0.0001),
              loss="categorical_crossentropy",
              metrics=['accuracy'])
history = model.fit(newTrainData, newTrainTarget, batch_size
=32,epochs=50, shuffle=True,
                    validation_data=(newValidData, newValidTarget))
```

2.2 Model Training

At the end of 50 epochs, the neural network had a 0.9277 Test Accuracy and 0.4401 Test Loss.

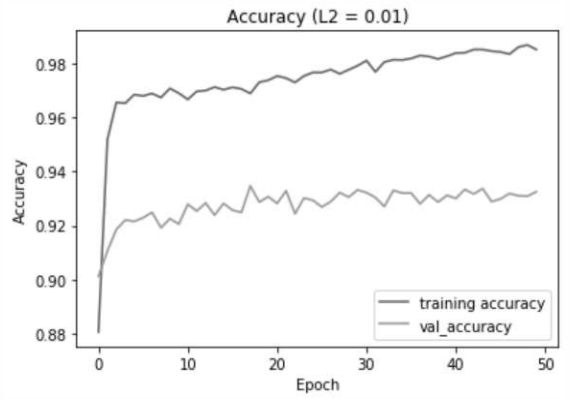


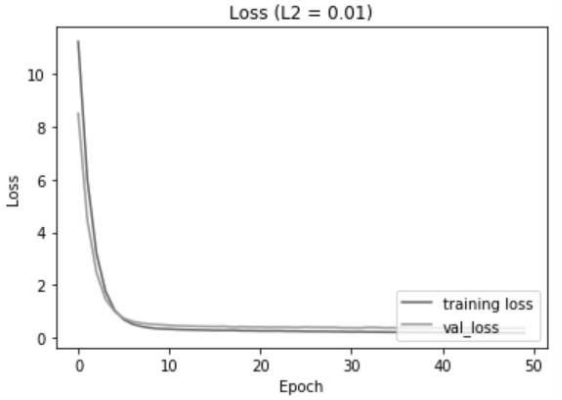
2.3 Hyperparameter Investigation

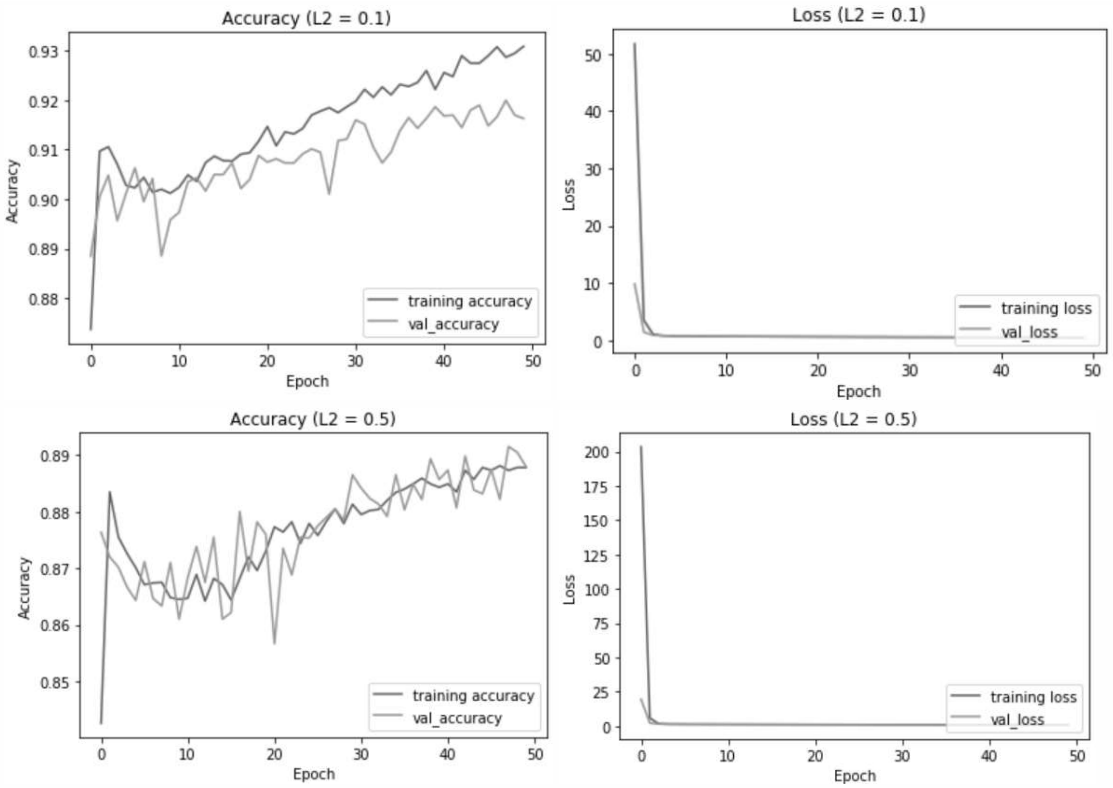
2.3.1 L2 Regularization

```
from tensorflow.keras import datasets, layers, models, optimizers,
regularizers

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), padding="same",strides=1,
activation='relu', use bias= True, bias initializer='zeros',
kernel initializer=initializers.glorot normal(seed=14), input shape=(28,
28, 1)))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(784, activation='relu', kernel regularizer =
regularizers.l2(0.01)))
model.add(layers.Dense(10, activation='softmax', kernel regularizer =
regularizers.l2(0.01)))

model.compile(optimizer=optimizers.Adam(learning rate=0.0001),
              loss="categorical crossentropy",
              metrics=['accuracy'])
history = model.fit(newTrainData, newTrainTarget, batch size =32,epochs=50,
shuffle=True,
                    validation data=(newValidData, newValidTarget))
```

λ	Training Acc	Training Loss	Valid Acc	Valid Loss	Test Acc	Test Loss
0.01	0.9852	0.1882	0.9325	0.3671	0.9269	0.3849
0.1	0.9309	0.4840	0.9163	0.5207	0.9203	0.5215
0.5	0.8878	0.9251	0.888	0.9438	0.8943	0.9298







Through observation of the plots above,lower regularization gave us a higher validation and test accuracy.

2.3.2 Dropout

```
from tensorflow.keras import datasets, layers, models, optimizers,
regularizers

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), padding="same",strides=1,
activation='relu', use_bias= True, bias_initializer='zeros',
kernel_initializer=initializers.glorot_normal(seed=14),
input_shape=(28, 28, 1)))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(784, activation='relu'))
#add dropout
model.add(layers.Dropout(0.9))
model.add(layers.Dense(10, activation='softmax'))

model.compile(optimizer=optimizers.Adam(learning_rate=0.0001),
              loss="categorical_crossentropy",
              metrics=['accuracy'])
history = model.fit(newTrainData, newTrainTarget, batch_size
=32,epochs=50, shuffle=True,
                    validation_data=(newValidData, newValidTarget))
```

The neural network performed better when the drop rate was high. Both the Validation and Test accuracy were highest when the dropout rate was p = 0.9.

p	Training Acc	Training Loss	Valid Acc	Valid Loss	Test Acc	Test Loss
0.9	0.9652	0.1076	0.9381	0.2698	0.9369	0.2731
0.75	0.9933	0.0258	0.9375	0.3692	0.9325	0.3905
0.5	0.9977	0.0094	0.9335	0.4470	0.9306	0.4787