ECE421 - Winter 2022 Assignment 1: Logistic Regression

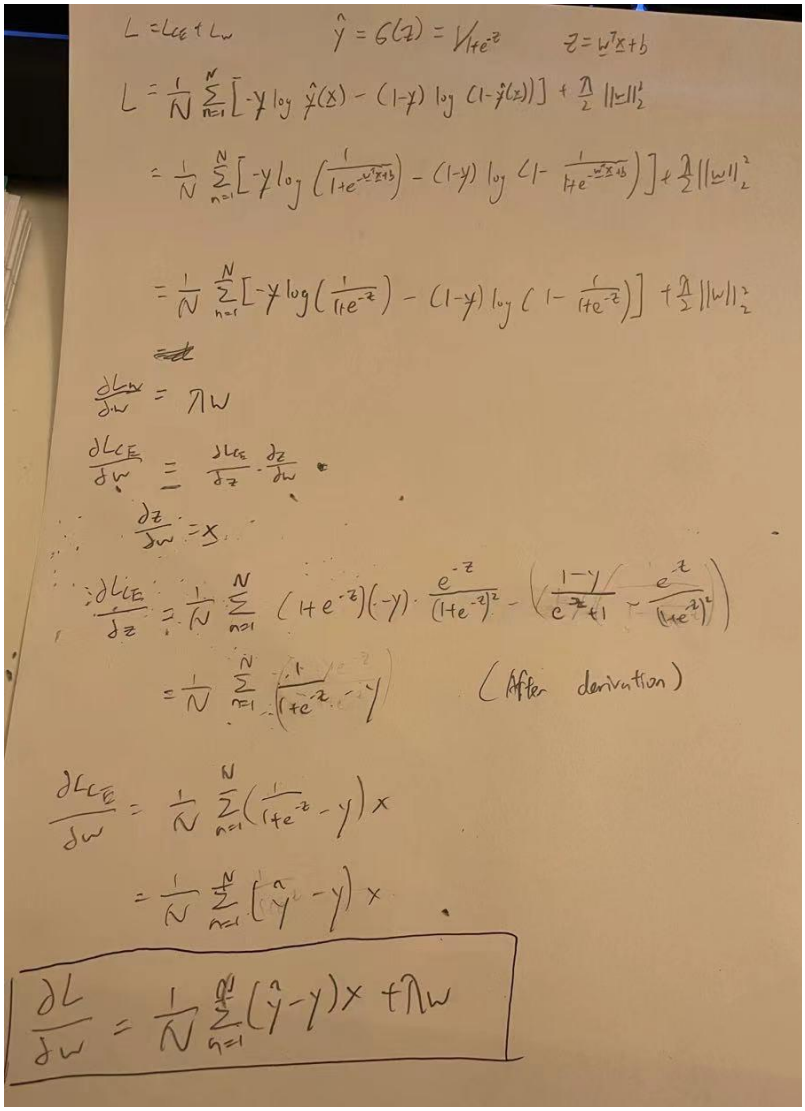
Raymond Hong 1003942629

Code written in Google Colab:
https://colab.research.google.com/drive/1TSWRr8PgGJYnP9pQgD__pDBXVEocbMb9#scrollTo=8O0cwfovkWt5

Part 1 Logistic Regression with Numpy

1. Gradient Derivation

- I converted y_hat to its logit expression and used a chain rule to compute the gradient of the cross entropy loss. The gradient of the regularization term is straightforward. Adding the two gradients give us the gradients of the total loss functions. Below is the steps of derivation and a snippet of the python code.



```python
def loss(w, b, x, y, reg):

    # logit
    z = np.matmul(x, w) + b

    # Probability Prediction
    y_hat = 1/(1 + np.exp(-z))

    # Cross Entropy Loss
    L_CE = -np.mean(y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat))

    # Regularization Term
    L_W = 0.5 * reg * (np.linalg.norm(w) ** 2)

    return L_W + L_CE

def grad_loss(w, b, x, y, reg):
    n = np.shape(y)[0]

    # logit
    z = np.matmul(x, w) + b

    # Probability Prediction
    y_hat = 1/(1 + np.exp(-z))

    # gradient of the loss w.r.t. w
    grad_w = (np.matmul(x.T, y_hat - y))/n + reg * w

    # gradient of the loss w.r.t. bias
    grad_b = np.sum(y_hat - y)/n

    return grad_w, grad_b
```

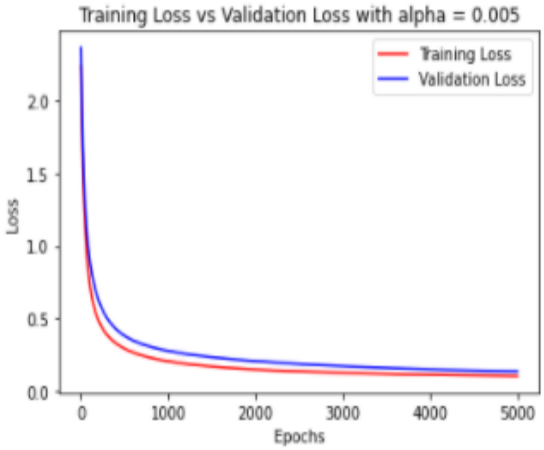
2. Gradient Descent Implementation

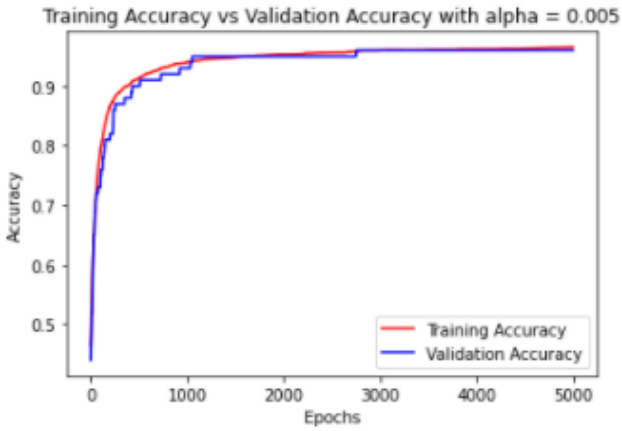
- For each epoch, I calculated the gradient loss and updated the weight and bias in response to the learning rate. I find the losses and accuracies for different sets of data at each epoch/iteration. When the error tolerance is smaller than the value we set, we end the for loop and return the data at that epoch. If the error is never smaller than the tolerance, we will just terminate when the for loop ends and this is determined by the number of iterations set.

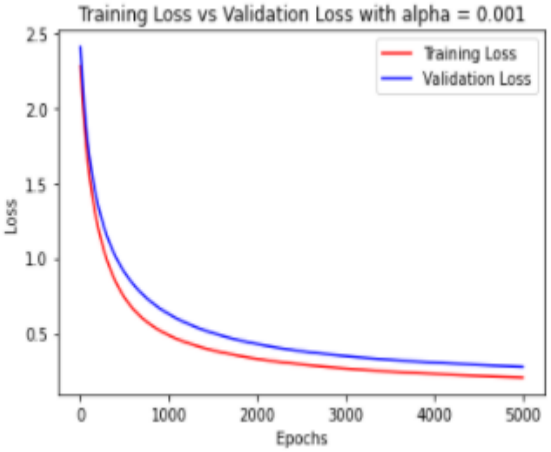
2.Gradient Descent Implementation

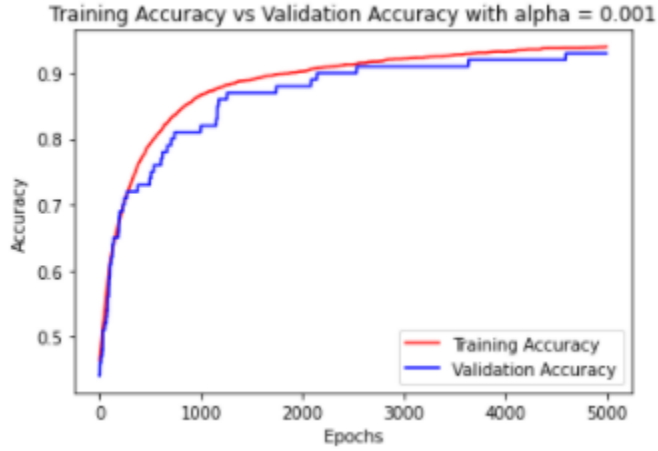
```
[25] def accuracy(w,b,x,y):
        z = np.matmul(x, w) + b

        y_hat = 1/(1 + np.exp(-z))
        return np.sum((y_hat >= 0.5) == y)/np.shape(y)[0]


    def grad_descent(w, b, trainingData, trainingLabels, validationData, validationLabels, testingData, testingLabels, alpha, epochs, reg, error_tol):
        training_loss, validation_loss, testing_loss, training_acc, validation_acc, testing_acc  = [], [], [], [], [], []

        for i in range(epochs):
            # Calculate gradient
            grad_w, grad_b = grad_loss(w, b, trainingData, trainingLabels, reg)
            w_new = w - alpha * grad_w
            b_new = b - alpha * grad_b

            # Find losses
            training_loss.append(loss(w_new,b_new,trainingData,trainingLabels,reg))
            validation_loss.append(loss(w_new,b_new,validationData,validationLabels,reg))
            testing_loss.append(loss(w_new,b_new,testingData,testingLabels,reg))

            # Find accuracies
            training_acc.append(accuracy(w,b,trainingData, trainingLabels))
            validation_acc.append(accuracy(w,b,validationData, validationLabels))
            testing_acc.append(accuracy(w,b,testingData, testingLabels))

            # Check if error is small enough
            if  np.linalg.norm(w_new - w) < error_tol:
                break

            # Update w and b
            w = w_new
            b = b_new

        return w_new, b_new, training_loss, validation_loss, testing_loss, training_acc, validation_acc, testing_acc
```

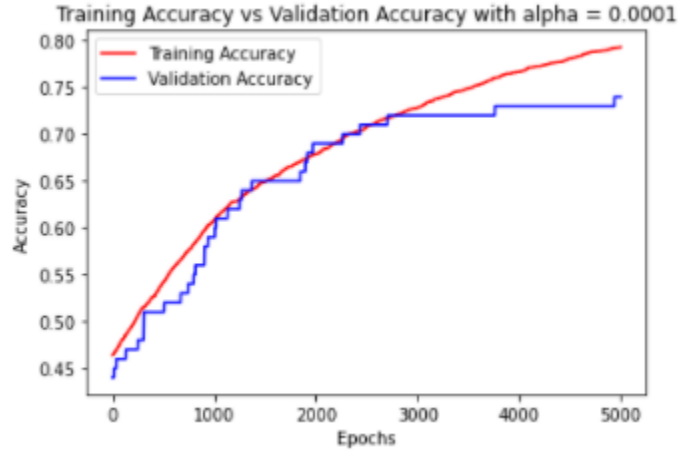
3. Tuning Learning Rate









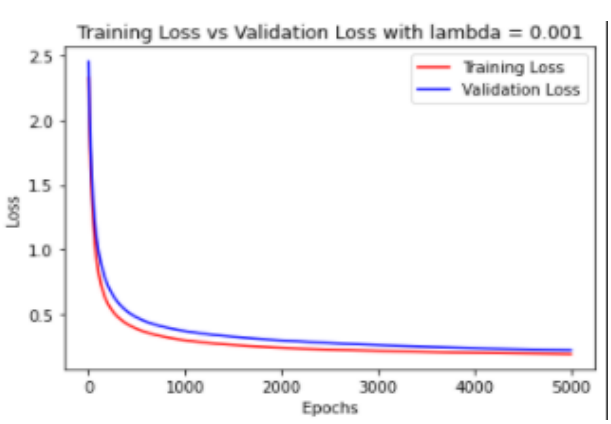


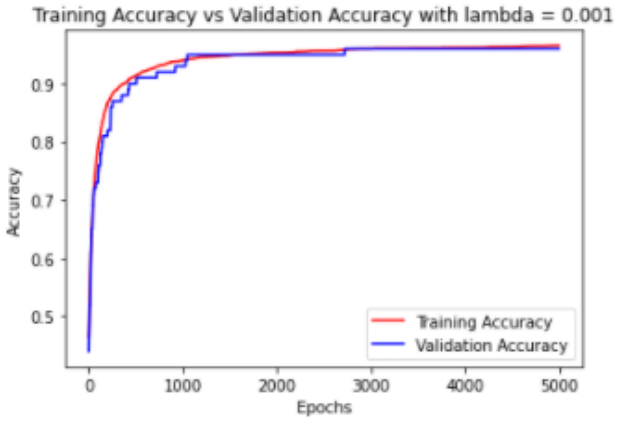
When learning rate = 0.005, the accuracy is the best. This makes sense because we are updating the weight using w_new = w - alpha * grad_w. As the learning rate gets bigger, loss decreases more drastically after each epoch. The accuracy also converges to a better percentage with the increasing learning rate.

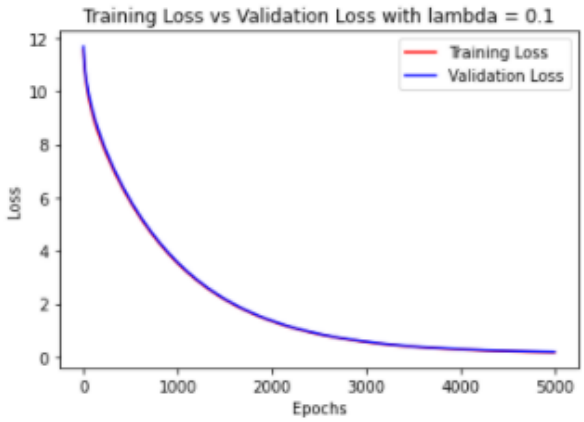
The training accuracy is **0.9654285714285714** and the validation accuracy is **0.96** when the learning rate is **0.005.** Below is a table of accuracies with other learning rates.

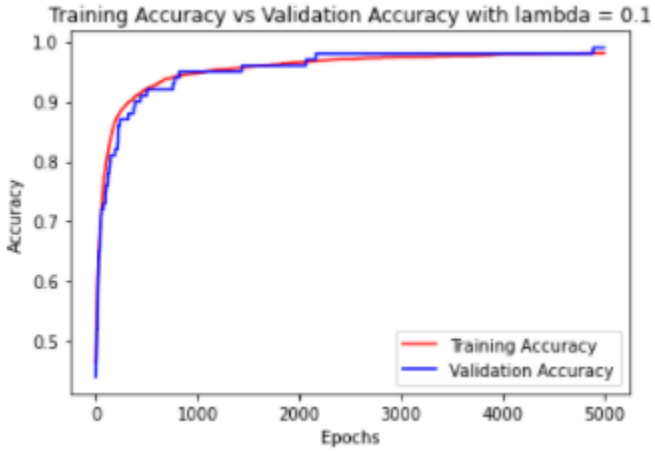
Lambda = 0	Training Accuracy	Validation Accuracy
Learning_rate = 0.005	0.9654285714285714	0.96
Learning_rate = 0.001	0.94	0.93
Learning_rate = 0.0001	0.7931428571428571	0.74

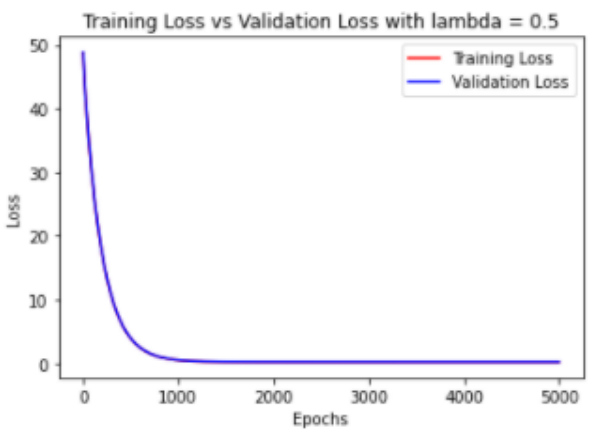
4. Tuning Regularization Parameter

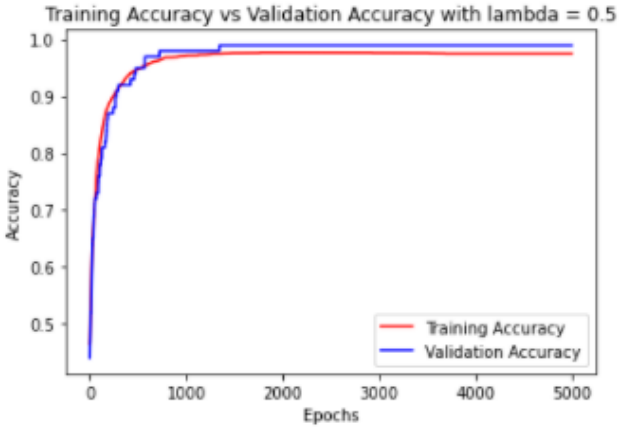












When regularization parameter = 0.1, the accuracy is the best. The training accuracy is **0.9808571428571429** and the validation accuracy is **0.99** when the regularization parameter is **0.1.** Below is a table of accuracies with other regularization parameters.

Learning_rate = 0.005	**Training Accuracy**	**Validation Accuracy**
Lambda = 0.001	**0.9657142857142857**	**0.96**
Lambda = 0.1	**0.9808571428571429**	**0.99**
Lambda = 0.5	**0.9754285714285714**	**0.99**

```
trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()

# get image size
img_size = np.shape(trainData)[1]* np.shape(trainData)[2]

# flatten images
trainData = trainData.reshape(np.shape(trainData)[0], img_size)
validData = validData.reshape(np.shape(validData)[0], img_size)
testData  = testData.reshape(np.shape(testData)[0], img_size)

# initialize
epochs = 5000
error_tol = 1e-7
w = np.random.normal(0, 0.5, (np.shape(trainData)[1],1))
b = 0


alphas = [0.005, 0.001, 0.0001]
lambdas = [0.001, 0.1, 0.5]
```

```
for alpha in alphas:
    new_w, new_b, train_loss, valid_loss, test_loss, train_acc, valid_acc, test_acc = grad_descent(w, b, trainData, trainTarget, validData, validTarget, testData, testTarget, alpha, epochs, 0, error_tol)
    plt.figure()
    plt.plot(range(epochs), train_loss, 'r-')
    plt.plot(range(epochs), valid_loss, 'b-')
    plt.legend(["Training Loss", "Validation Loss"])
    plt.xlabel('Epochs')
    plt.ylabel("Loss")
    plt.title("Training Loss vs Validation Loss with alpha = " + str(alpha))


    plt.figure()
    plt.plot(range(epochs), train_acc, 'r-')
    plt.plot(range(epochs), valid_acc, 'b-')
    plt.legend(["Training Accuracy", "Validation Accuracy"])
    plt.xlabel('Epochs')
    plt.ylabel("Accuracy")
    plt.title("Training Accuracy vs Validation Accuracy with alpha = " + str(alpha))
    # print(valid_acc[-1])
    # print(train_acc[-1])
```

```
for lam in lambdas:
    new_w, new_b, train_loss, valid_loss, test_loss, train_acc, valid_acc, test_acc = grad_descent(w, b, trainData, trainTarget, validData, validTarget, testData, testTarget, 0.005, epochs, lam, error_tol)
    plt.figure()
    plt.plot(range(epochs), train_loss, 'r-')
    plt.plot(range(epochs), valid_loss, 'b-')
    plt.legend(["Training Loss", "Validation Loss"])
    plt.xlabel('Epochs')
    plt.ylabel("Loss")
    plt.title("Training Loss vs Validation Loss with lambda = " + str(lam))


    plt.figure()
    plt.plot(range(epochs), train_acc, 'r-')
    plt.plot(range(epochs), valid_acc, 'b-')
    plt.legend(["Training Accuracy", "Validation Accuracy"])
    plt.xlabel('Epochs')
    plt.ylabel("Accuracy")
    plt.title("Training Accuracy vs Validation Accuracy with lambda = " + str(lam))
    # print(valid_acc[-1])
    # print(train_acc[-1])
```

Part 2 Logistic Regression in TensorFlow

1. Building the Computational Graph

Here is the buildGraph function that I came up with. It takes a few input parameters such as alpha, beta1, beta2 and epsilon and it outputs weight, bias, predicted labels, real labels, the loss

and the optimizer as well as the regularization parameter.

```
def buildGraph (beta1, beta2, epsilon, alpha):
  w = tf.Variable(tf.truncated_normal([784,1], mean = 0, stddev = 0.5, dtype = tf.float32))
  b = tf.Variable(tf.zeros(1))

  x = tf.placeholder(tf.float32, [None, 784])
  y = tf.placeholder(tf.float32, [None, 1])
  reg = tf.placeholder(tf.float32)

  logits = tf.matmul(x, w) + b

  loss = tf.losses.sigmoid_cross_entropy(multi_class_labels= y, logits = logits) + reg * tf.nn.l2_loss(w)

  optimizer = tf.train.AdamOptimizer(alpha, beta1, beta2, epsilon).minimize(loss)

  return w, b, x, y ,reg, loss, optimizer
```

I assign default values as following: (regularization parameter is set as 0)

```
[9] # initialize variables
    batch_size = 500
    beta1 = 0.9
    beta2 = 0.999
    epsilon = 1e-7
    alpha = 0.001
    epochs = 700
```

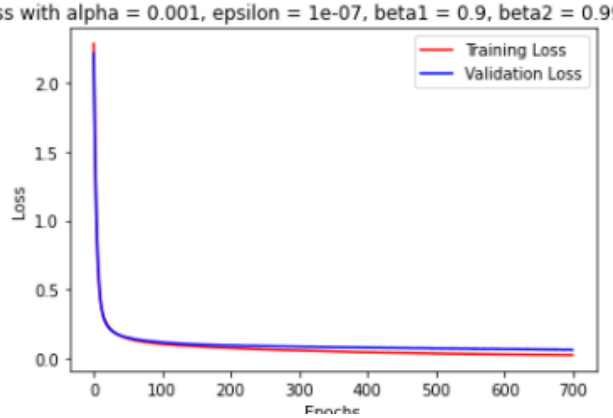
2. Implementing SGD

Train the data using SGD algorithm with a minibatch size of 500 over 700 epochs. The total number of batches is calculated by dividing the number of training instances over the minibatch size. I iterate over the number of batches with the adam optimizer. In each epoch, I also reshuffle the training data and start sampling from the beginning again. At each iteration, we select samples uniformly and randomly. This means that we can do more iterations than the normal gradient descent method. Often there is redundancy in big datasets, with SGD, we do not consider all data points in each step. The training and validation loss/accuracy is stored at

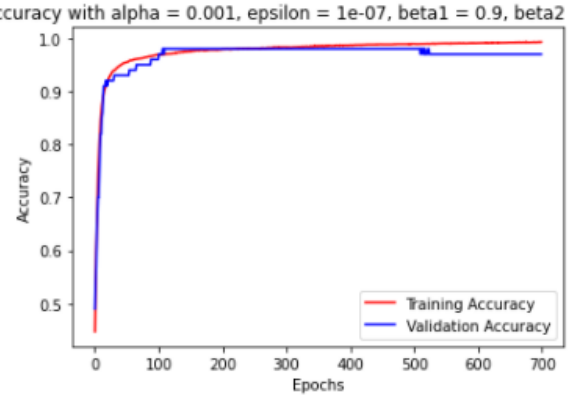
each epoch and displayed below.

```
def SGD(batchSize, beta1, beta2, epsilon,alpha , epochs):
    trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
    tf.set_random_seed(421)
    img_size =  np.shape(trainData)[1] * np.shape(trainData)[2]
    trainData = trainData.reshape(np.shape(trainData)[0], img_size)
    validData = validData.reshape(np.shape(validData)[0],  img_size)
    testData  = testData.reshape(np.shape(testData)[0],  img_size)

    w, b, x, y, reg, loss, adam_op = buildGraph(beta1, beta2, epsilon, alpha)
    train_loss, valid_loss, test_loss, train_acc, valid_acc, test_acc  = [], [], [], [], [], []

    num_batches = np.shape(trainData)[0] // batchSize
    init = tf.global_variables_initializer()

    with tf.Session() as sess:
      sess.run(init)

      for i in range(epochs):
        index = np.arange(trainData.shape[0])
        np.random.shuffle(index)
        trainData = trainData [index, :]
        trainTarget = trainTarget[index]

        for j in range(num_batches):
          x_batch = trainData[j * batchSize: (j+1) * batchSize, :]
          y_batch = trainTarget[j * batchSize: (j+1) * batchSize, :]
          _ , new_loss, new_w ,new_b = sess.run([adam_op, loss, w, b], feed_dict = {x: x_batch, y: y_batch, reg: 0})

        train_loss.append(sess.run(loss, feed_dict = {x: trainData, y: trainTarget, reg: 0}))
        valid_loss.append(sess.run(loss, feed_dict = {x: validData, y: validTarget, reg: 0}))
        test_loss.append(sess.run(loss, feed_dict = {x: testData, y: testTarget, reg: 0}))


        train_acc.append(accuracy(new_w, new_b, trainData, trainTarget))
        valid_acc.append(accuracy(new_w, new_b, validData, validTarget))
        test_acc.append(accuracy(new_w, new_b, testData, testTarget))

    return train_loss, valid_loss, test_loss, train_acc, valid_acc, test_acc
```

Training Loss vs Validation Loss with alpha = 0.001, epsilon = 1e-07, beta1 = 0.9, beta2 = 0.999, batch size = 500, epochs = 700



Training Accuracy vs Validation Accuracy with alpha = 0.001, epsilon = 1e-07, beta1 = 0.9, beta2 = 0.999, batch size = 500, epochs = 700



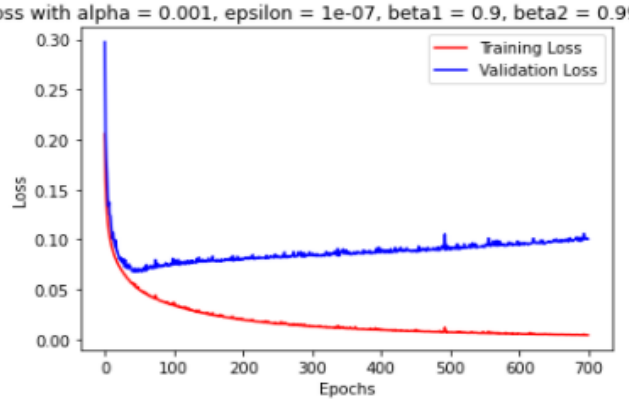
Training accuracy is **0.9931428571428571** and validation accuracy is **0.97** after training.

3. Batch Size Investigation

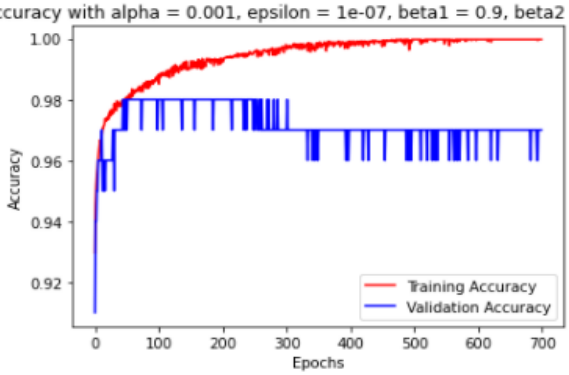
I train the data with 3 different batch sizes: 100, 700, 1750 with the same model as above. Below are the plots generated for each batch size:

	Training Accuracy	**Validation Accuracy**
Batch Size= 100	**0.9997142857142857**	**0.97**
Batch Size= 170	**0.9862857142857143**	**0.97**
Batch Size= 1750	**0.9794285714285714**	**0.98**

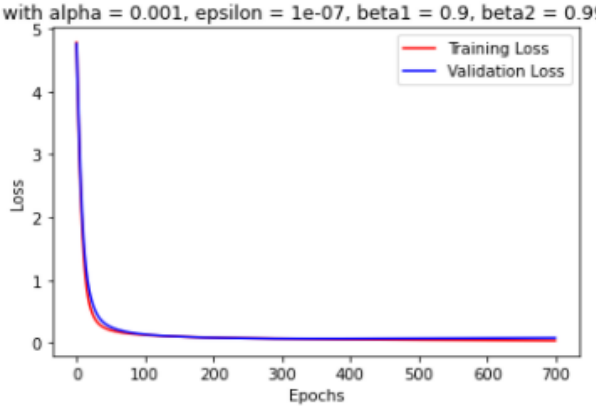
Training Loss vs Validation Loss with alpha = 0.001, epsilon = 1e-07, beta1 = 0.9, beta2 = 0.999, batch size = 100, epochs = 700



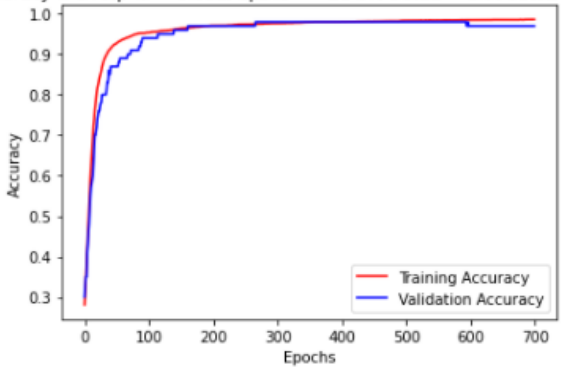
Training Accuracy vs Validation Accuracy with alpha = 0.001, epsilon = 1e-07, beta1 = 0.9, beta2 = 0.999, batch size = 100, epochs = 700



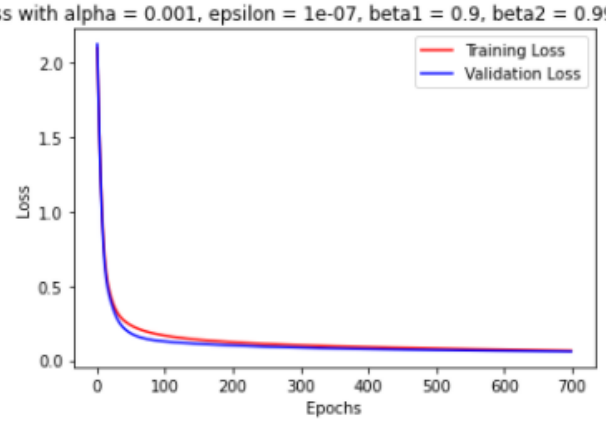
Training Loss vs Validation Loss with alpha = 0.001, epsilon = 1e-07, beta1 = 0.9, beta2 = 0.999, batch size = 700, epochs = 700



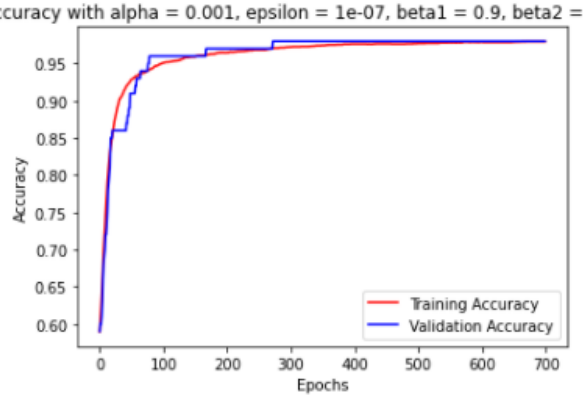
Training Accuracy vs Validation Accuracy with alpha = 0.001, epsilon = 1e-07, beta1 = 0.9, beta2 = 0.999, batch size = 700, epochs = 700



Training Loss vs Validation Loss with alpha = 0.001, epsilon = 1e-07, beta1 = 0.9, beta2 = 0.999, batch size = 1750, epochs = 700



Training Accuracy vs Validation Accuracy with alpha = 0.001, epsilon = 1e-07, beta1 = 0.9, beta2 = 0.999, batch size = 1750, epochs = 700

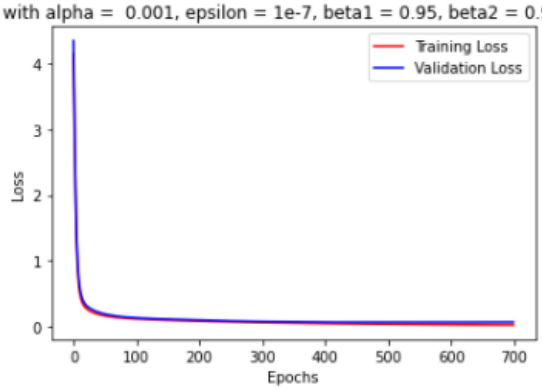


As we can see from the graphs and the data, the larger the batch size the lower the training accuracy. However, the training accuracy is not significantly lower when the batch size increases. We can see the curves are more smooth when the batch size is bigger and this means there is less noise. With a smaller batch size, we have a more noisy estimate of the true gradient at each iteration.

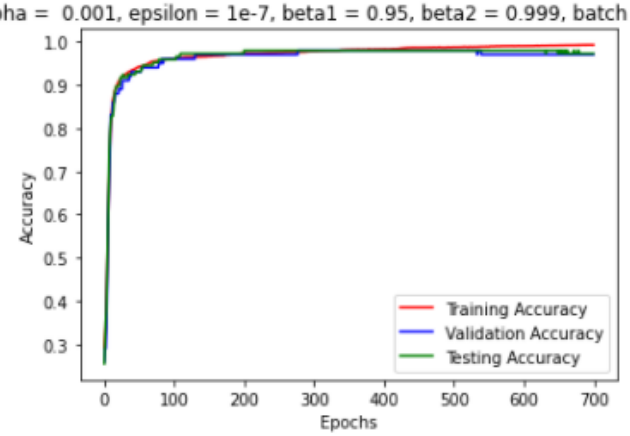
4. Hyperparameter Investigation

	Training Accuracy	**Validation Accuracy**	**Testing Accuracy**
beta1= 0.95	**0.9925714285714285**	**0.97**	**0.9724137931034482**
beta1= 0.99	**0.9948571428571429**	**0.96**	**0.9793103448275862**

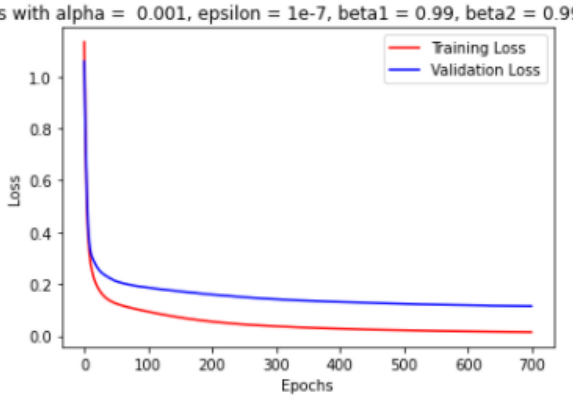
Training Loss vs Validation Loss with alpha = 0.001, epsilon = 1e-7, beta1 = 0.95, beta2 = 0.999, batch size = 500, epochs = 700



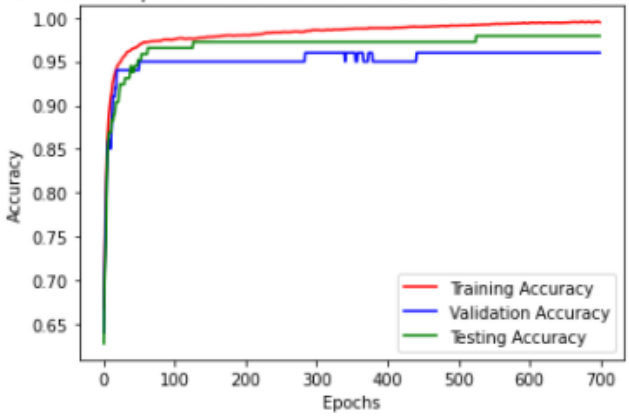
Accuracies with alpha = 0.001, epsilon = 1e-7, beta1 = 0.95, beta2 = 0.999, batch size = 500, epochs = 700



Training Loss vs Validation Loss with alpha = 0.001, epsilon = 1e-7, beta1 = 0.99, beta2 = 0.999, batch size = 500, epochs = 700

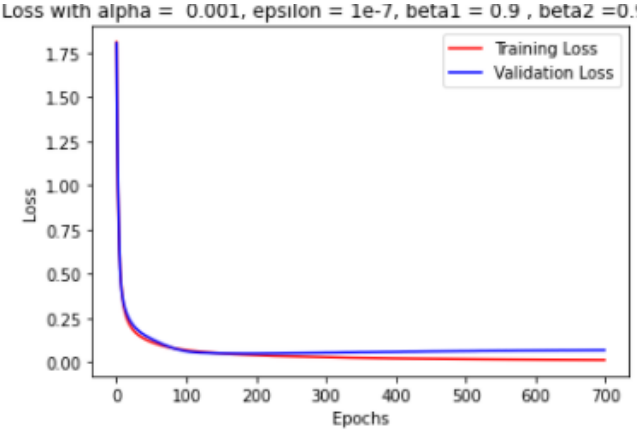


Accuracies with alpha = 0.001, epsilon = 1e-7, beta1 = 0.99, beta2 = 0.999, batch size = 500, epochs = 700

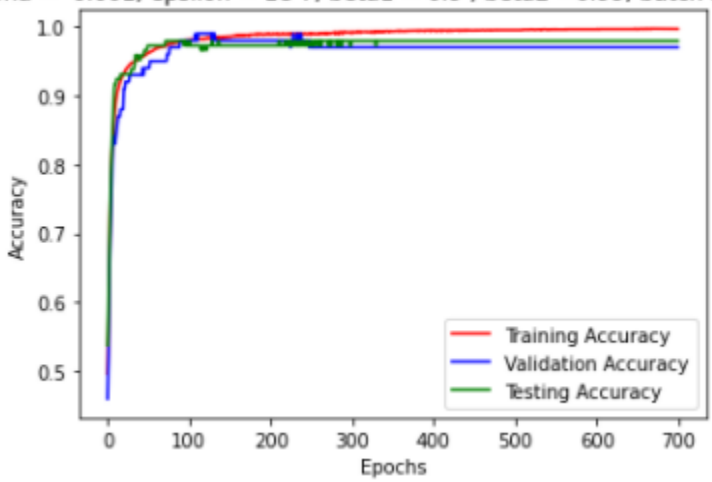


	Training Accuracy	Validation Accuracy	Testing Accuracy
beta2= 0.99	**0.9968571428571429**	**0.97**	**0.9793103448275862**
beta2= 0.9999	**0.9854285714285714**	**0.96**	**0.9793103448275862**

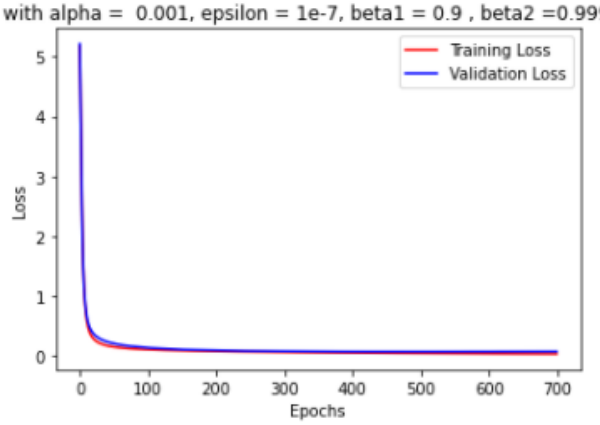
Training Loss vs Validation Loss with alpha = 0.001, epsilon = 1e-7, beta1 = 0.9 , beta2 =0.99, batch size = 500, epochs = 700



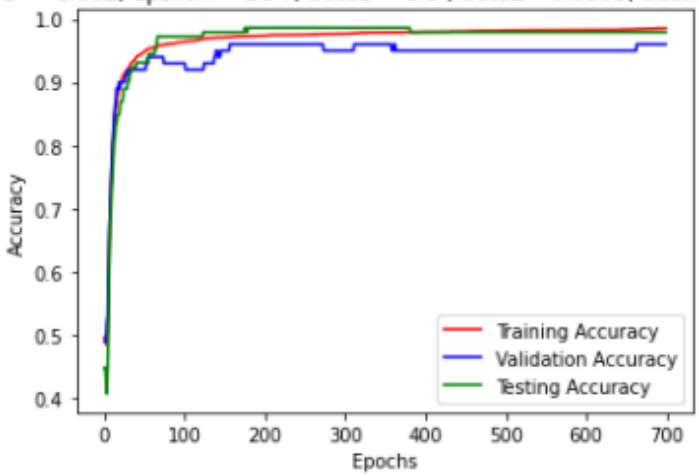
Accuracies with alpha = 0.001, epsilon = 1e-7, beta1 = 0.9 , beta2 =0.99, batch size = 500, epochs = 700



Training Loss vs Validation Loss with alpha = 0.001, epsilon = 1e-7, beta1 = 0.9 , beta2 =0.9999, batch size = 500, epochs = 700

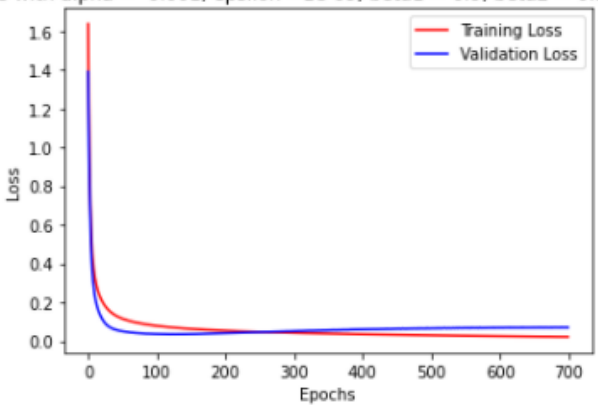


Accuracies with alpha = 0.001, epsilon = 1e-7, beta1 = 0.9 , beta2 =0.9999, batch size = 500, epochs = 700

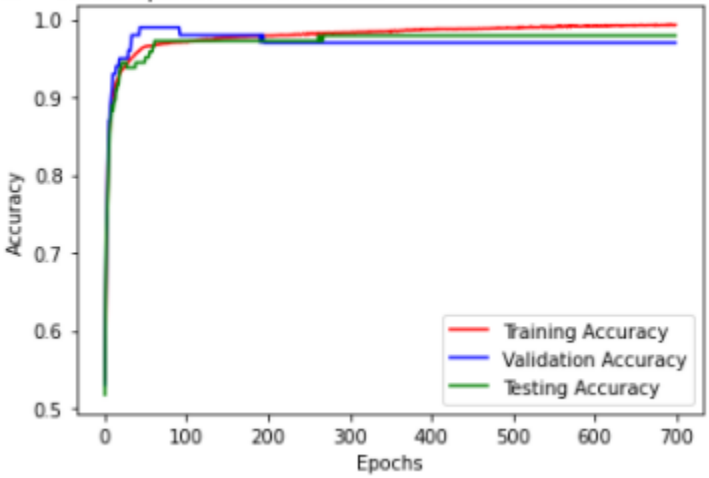


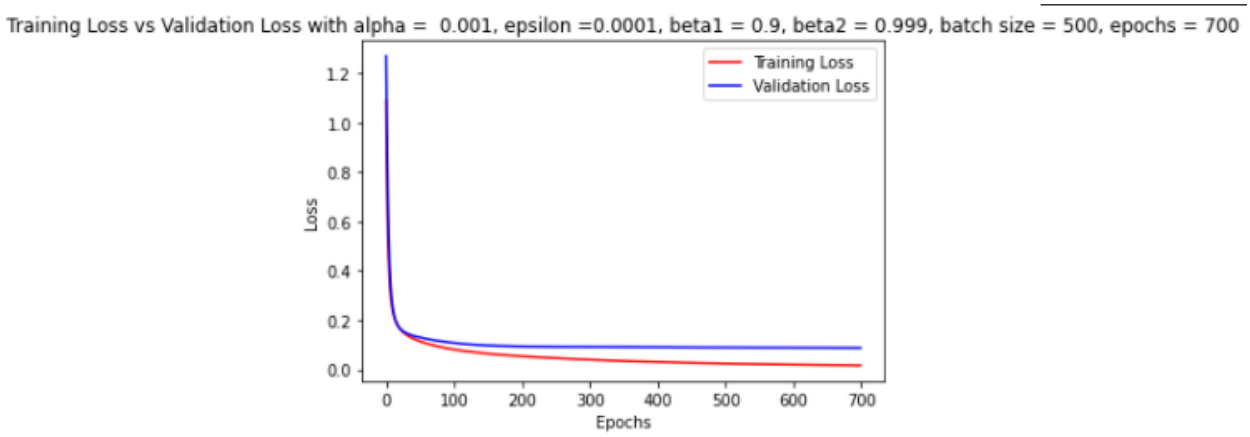
	Training Accuracy	Validation Accuracy	Testing Accuracy
epsilon= 1e-9	**0.9934285714285714**	**0.97**	**0.9793103448275862**
epsilon= 1e-4	**0.9948571428571429**	**0.97**	**0.9862068965517241**

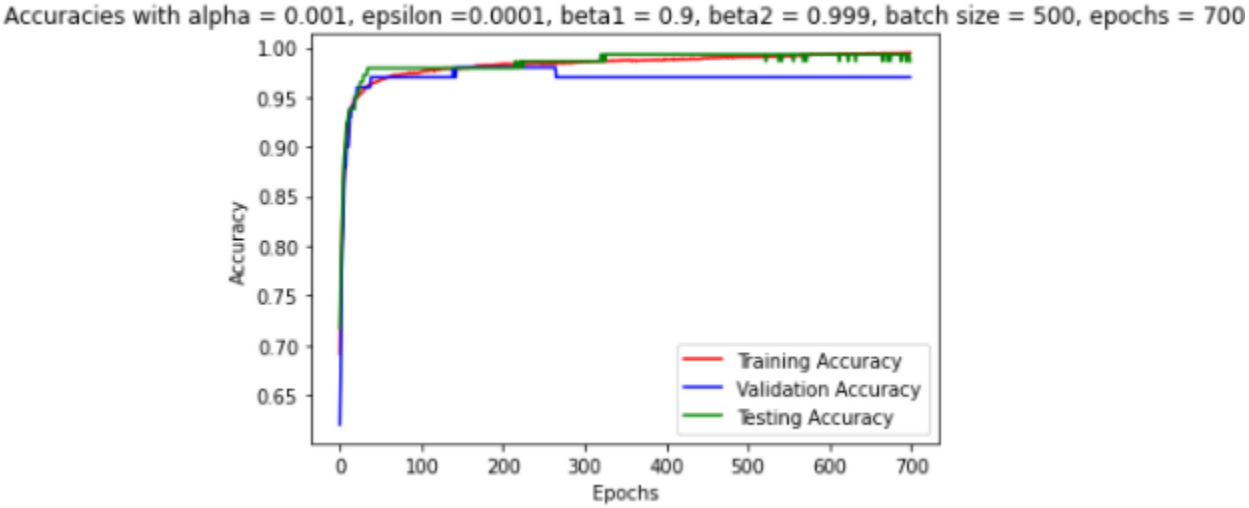
Training Loss vs Validation Loss with alpha = 0.001, epsilon =1e-09, beta1 = 0.9, beta2 = 0.999, batch size = 500, epochs = 700



Accuracies with alpha = 0.001, epsilon =1e-09, beta1 = 0.9, beta2 = 0.999, batch size = 500, epochs = 700







For beta1, I suggest that 0.95 is the better value as the testing accuracy matches better with the training and validation accuracy with less noise. Even though when beta1 = 0.99, the testing accuracy is slightly better, the accuracy curves are less smooth (due to noise).

For beta2, I suggest using 0.99. As beta 2 increases, the training accuracy decreases and the curves are also less smooth (due to noise).

For epsilon, I suggest using 1e-4 as all accuracies are higher with this value than the other one.

5. Comparison against Batch GD

The Adam Optimizer performs better than the Batch Gradient Descent because it can reach a lower loss and a higher accuracy in less epochs than Batch GD(3500 vs 700). However, there are more noises with the various hyperparameters of Adam Optimizer. This can be reduced by adjusting different parameters. Since Adam Optimizer has more variables associated with it, it is more complicated. On the other side of the spectrum, the Adam Optimizer can also be more detailed in the training model adjustments with these hyperparameters.