ECE421 Assignment 1

Student	Ⓥ Valentina Manferrari
utorID	1004796615

PART 1

1. Loss Function and Gradient

$$\mathcal{L} = \mathcal{L}_{CE} + \mathcal{L}_w =$$
$$= \frac{1}{N}\sum_{n=1}^{N}[-y^{(n)}log\hat{y}(x^{(n)}) - (1-y^{(n)})log(1-\hat{y}(x^{(n)}))] + \frac{\lambda}{2}||\mathbf{w}||_2^2$$

where $\hat{y}(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b)$ and $\sigma = \frac{1}{1+e^{-z}}$

```
def loss(W, b, x, y, reg):
  # the np.matmul function is used for matrix multiplication
  z = np.matmul(x,W) + b
  Y = sigmoid(z)

  # total_loss = cross-entropy_loss + regularizarion_term
  # np.sum return the sum of array elements over a given axis
  # np.shape function returns a tuple with each index having
  #the number of corresponding elements (we only need index 0)
  loss_CE = np.sum(-(y*np.log(Y))-(1-y)*np.log(1-Y))/np.shape(y)[0]
  loss_w = reg/2 * np.sum(W*W) # reg = lambda

  total_loss = loss_CE + loss_w
  return total_loss

# Defining the sigmoid as an helper function
def sigmoid(x):
  return 1 / (1 + np.exp(-x))
```

In order to find the gradient of the loss function we proceed by applying the chain rule as follows:

$$\frac{\partial\mathcal{L}}{\partial w} = \frac{\partial\mathcal{L}}{\partial\hat{y}(\mathbf{x})}\cdot\frac{\partial\hat{y}(\mathbf{x})}{\partial\mathbf{x}}\cdot\frac{\partial\mathbf{x}}{\partial w} = \frac{1}{N}\mathbf{x}^T(\hat{y}-y) + \lambda\mathbf{w}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{N}(\hat{y} - y)$$

```
def grad_loss(W, b, x, y, reg):
  z = np.matmul(x,W) + b
  Y = sigmoid(z)

  #calculating the loss gradient with respect to the weights
  grad_w = np.matmul(np.transpose(x), (Y-y))/np.shape(y)[0]
  #calculating the loss gradient with respect to the bias
  grad_b = np.sum(Y-y)/np.shape(y)[0]

  return grad_w, grad_b # function returns a tuple with each value as an element
```

2. Gradient Descent Implementation

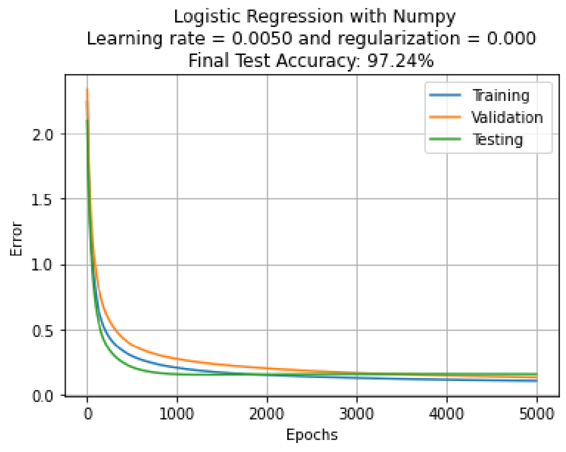
```
# grad_descent takes as arguments the weight vector W, the bias b, the data matrix x,
# the labels vector y, the learning rate alpha, the number of epochs,
# the lambda parameter (reg), error tolerance error_tol = 1*10^(-7)
# and extraInfo that contains the validation dataset paramenters
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol, extraInfo):

  # These variables are necessary for plotting the graphs
  train_err = loss(W, b, x, y, reg)
  val_err = loss(W, b, extraInfo[1], extraInfo[4], reg)
  test_err = loss(W, b, extraInfo[2], extraInfo[5], reg)
  train_acc = accuracy(W, b, x, y)
  val_acc = accuracy(W, b, extraInfo[1], extraInfo[4])
  test_acc = accuracy(W, b, extraInfo[2], extraInfo[5])
  # --- end of definition ---

  weights = W
  bias = b
  # Gradient Descent Algorithm
  for i in range(epochs-1):
    gradLoss_w_b = grad_loss(weights, bias, x, y, reg)
    updated_w = weights - alpha*gradLoss_w_b[0]
    updated_b = bias - alpha*gradLoss_w_b[1]
    magnitude = np.linalg.norm(updated_w - weights)
    if (magnitude < error_tol):
      return updated_w, updated_b
    else:
      weights = updated_w
      bias = updated_b
    # --- end of algorithm implementation ---

    # updating variables with new values every 10 epoches
    # this choice was made because sampling makes graph
    # plotting quicker but still accurate enough
    #if (i % 10 == 0):
    train_err = np.append(train_err, loss(weights, bias, x, y, reg))
    val_err = np.append(val_err, loss(weights, bias, extraInfo[1], extraInfo[4], reg))
```

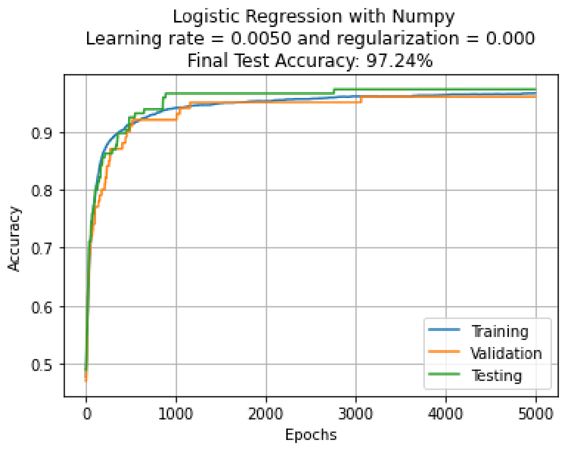
```
        test_err = np.append(test_err, loss(weights, bias, extraInfo[2], extraInfo[5], reg))
        train_acc = np.append(train_acc, accuracy(weights, bias, x, y))
        val_acc = np.append(val_acc, accuracy(weights, bias, extraInfo[1], extraInfo[4]))
        test_acc = np.append(test_acc, accuracy(weights, bias, extraInfo[2], extraInfo[5]))

    # plotting error and accuracy over epoches for training, validation and test datatsets
    plot(train_err, val_err, test_err, accuracy(weights, bias,
          extraInfo[2], extraInfo[5]), alpha, reg, "Error")
    plot(train_acc, val_acc, test_acc, accuracy(weights, bias,
          extraInfo[2], extraInfo[5]), alpha, reg, "Accuracy")

    return weights, bias
```

3. Tuning the Learning Rate

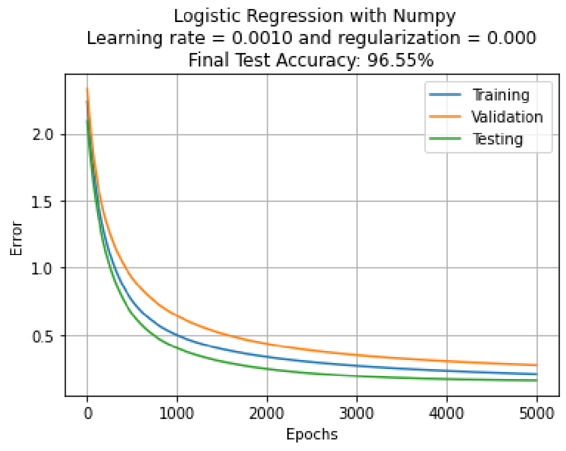
Results from testing my implementation of the Gradient Descent with 5000 epochs and $\lambda = 0$.

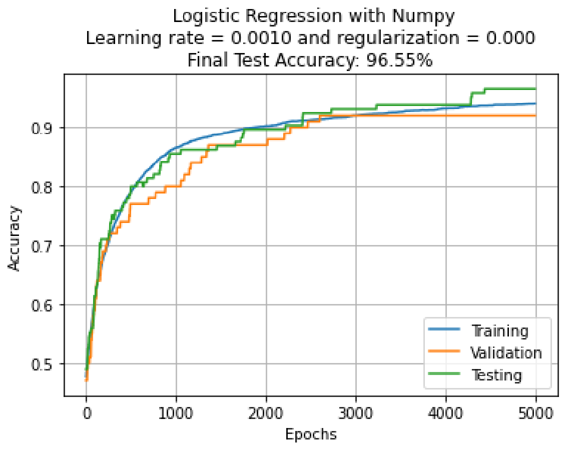
Learning rate = 0.005



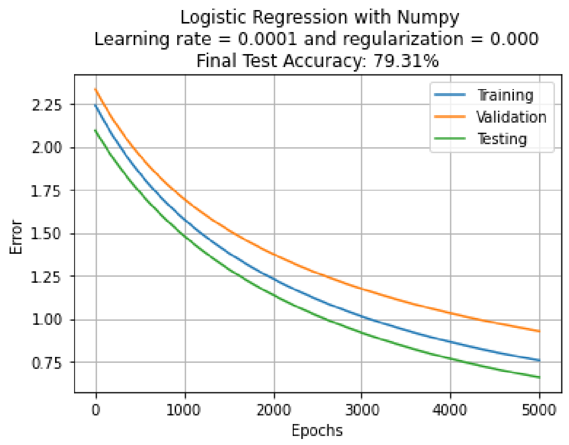


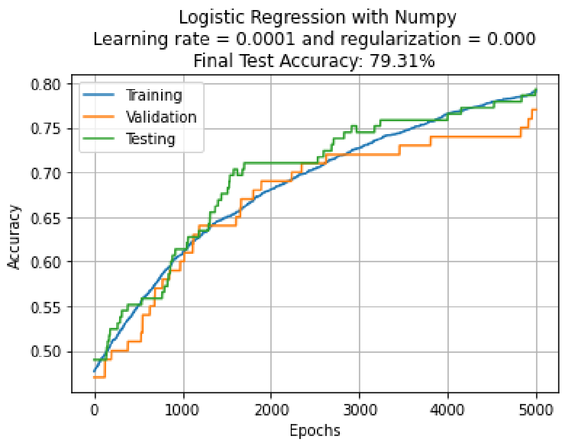
Learning rate = 0.001





Learning rate = 0.0001



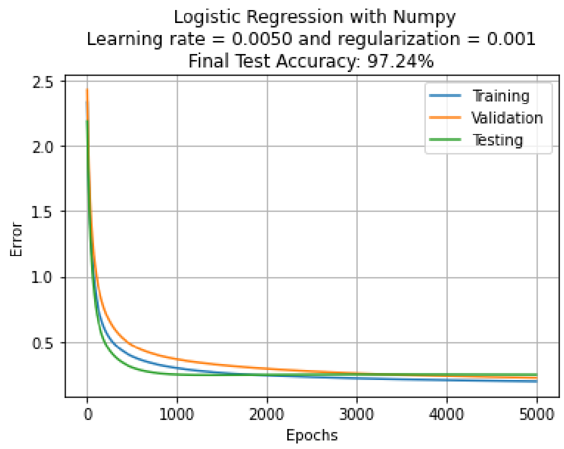


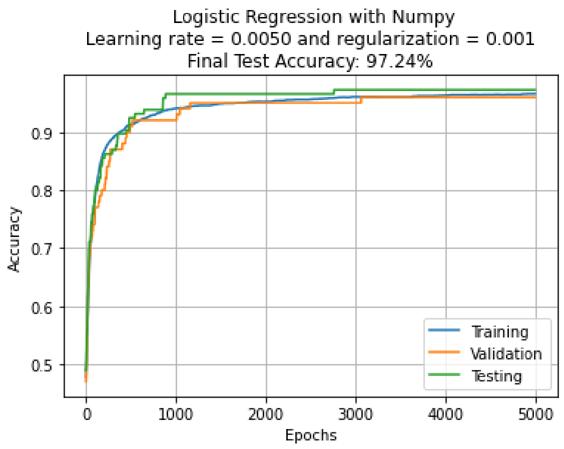
The choice of the learning rate value is critical for the training of the model: if it is too low the optimizer never progresses, and if it is too high it causes instability and the optimizer never converges. For this reason, it is important to find a value in between, a band of “just right” learning rates that successfully train. From the graphs above I would choose the learning rate value to be 0.005 since it leads to the highest testing dataset accuracy (97.24%). In addition, this learning rate produces the smallest loss and has a narrower "gap" between the losses of training, validating and testing datasets.

4. Generalization

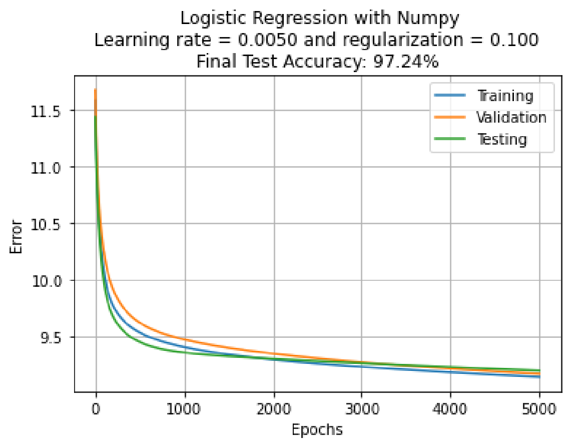
Results from testing my implementation of the Gradient Descent with 5000 epochs and $\alpha = 0$.

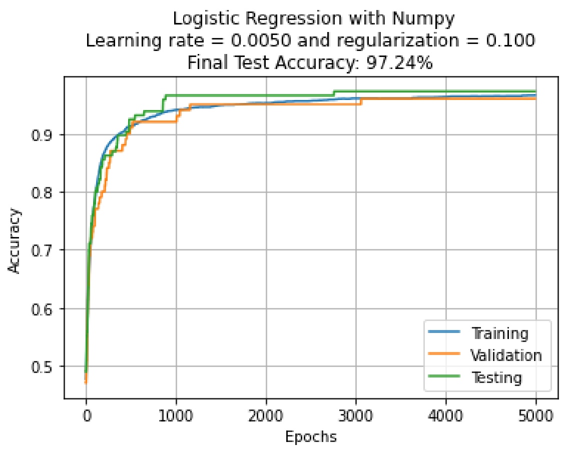
Regularization = 0.001



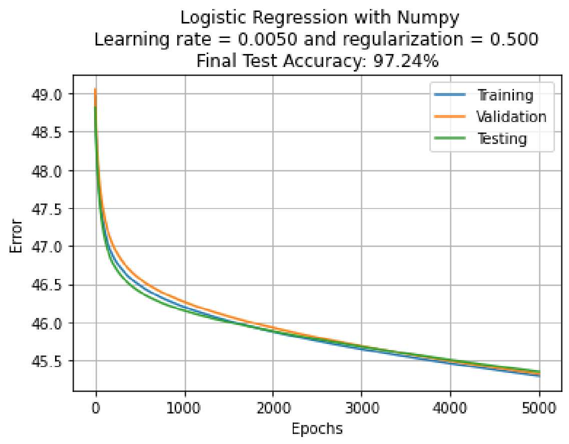


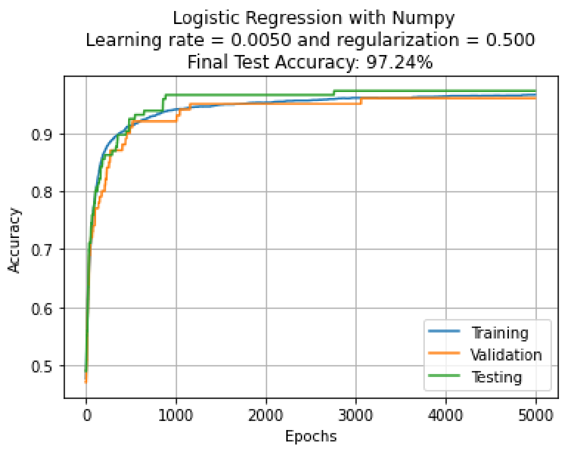
Regularization = 0.1





Regularization = 0.5





Regularization can be thought of as increasing the bias if our model suffers from high variance - it overfits the training data. Too much bias, however, will result in underfitting, causing the model to show a bad performance for both the training and test dataset. Since our goal in an unregularized model is to minimize the cost function, i.e. we want to find the feature weights that correspond to the global cost minimum, I think a good choice of regression value would be 0.001, reporting final test accuracy of 97.24%. It is clear that with the above-tested values the effects were very minimal and the final test accuracy stays the same, but, by testing increasingly higher regularization values I observed that the final test accuracy decreases quite quickly (overfit). For this reason, I think that tuning with regularization might require a larger range of values than those shown above.

PART 2

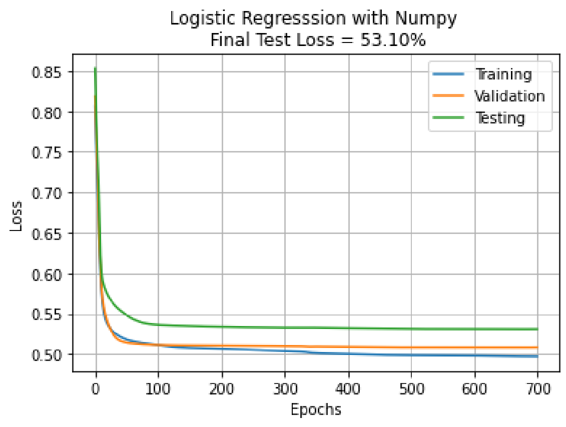
1. Building the Computational Graph

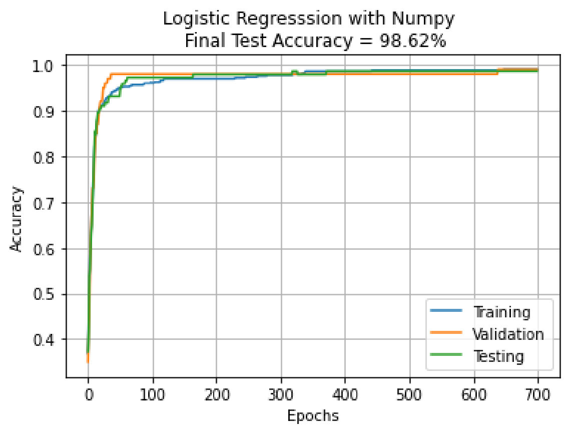
```
def buildGraph(beta1=None, beta2=None, epsilon=None, Lambda = 0, alpha=0.001,
               batch_size = 100, epochs = 700):
    tf.set_random_seed(421)
    tf.reset_default_graph()
    g = tf.Graph()
    with g.as_default():
        # Step A: defining the weight and bias tensor
        W = tf.Variable(tf.truncated_normal(shape=(784, 1),
                         mean=0.0, stddev=0.5, dtype=tf.float32))
        b = tf.Variable(tf.zeros(1))

        # Step B: ceating placehoders for data, labels and lambda
        train_data = tf.placeholder(tf.float32, shape=(batch_size, 784))
        train_label = tf.placeholder(tf.float32, shape=(batch_size, 1))
        val_data = tf.placeholder(tf.float32, shape=(100, 784))
        val_label = tf.placeholder(tf.int8, shape=(100, 1))
        test_data = tf.placeholder(tf.float32, shape=(145, 784))
        test_label = tf.placeholder(tf.int8, shape=(145, 1))
        reg = tf.placeholder(tf.float32, shape = (1))

        # Step C: calculating the loss tensor
        z_train = tf.matmul(train_data, W) + b
        train_prediction = tf.sigmoid(z_train)
        train_loss = tf.losses.sigmoid_cross_entropy(train_label, train_prediction)
        z_val = tf.matmul(val_data, W) + b
        valid_prediction = tf.sigmoid(z_val)
        valid_loss = tf.losses.sigmoid_cross_entropy(val_label, valid_prediction)
        z_test = tf.matmul(test_data,W) + b
        test_prediction = tf.sigmoid(z_test)
        test_loss = tf.losses.sigmoid_cross_entropy(test_label, test_prediction)

        # Step D: implementing the Adam optimizer to minimize the total loss
        ADAM = tf.train.AdamOptimizer(learning_rate=alpha, beta1=beta1, beta2=beta2,
                epsilon=epsilon).minimize(train_loss)
        data = [train_data, val_data, test_data]
        labels = [train_label, val_label, test_label]
        predictions = [train_prediction, valid_prediction, test_prediction]
        loss = [train_loss, valid_loss, test_loss]

        return W, b, data, labels, predictions, loss, g, ADAM
```

2. Implementing Stochastic Gradient Descent

```
def train_model_SGD(W, b, data, labels, predictions, loss, graph, ADAM,
        epsilon=0.0001, Lambda = 0, alpha=0.001, batch_size = 500, epochs = 700):
    train_prediction = predictions[0]
    valid_prediction = predictions[1]
    test_prediction = predictions[2]

    with tf.Session(graph=graph) as session:
        # Calculating the total number of batches required
        # by dividing the number of training istances by the minibatch size
        iterations = int(3500/batch_size)
        tf.global_variables_initializer().run()
        # Initializing arrays to store plot data
        training_loss, train_acc, validating_loss,
                valid_acc, testing_loss, test_acc = [], [], [], [], [], []

        for i in range(epochs):
            trainData, validData, testData,
                trainTarget, validTarget, testTarget = loadData()
            trainData = trainData.reshape((trainData.shape[0],
                            trainData.shape[1]*trainData.shape[2]))
            validData = validData.reshape((-1,validData.shape[1]*validData.shape[2]))
            testData = testData.reshape((-1,testData.shape[1]*testData.shape[2]))

            for j in range(iterations):
                X_batch = trainData[j*batch_size:(j+1)*batch_size,]
                Y_batch = trainTarget[j*batch_size:(j+1)*batch_size,]
                u, trained_W, trained_b, l, predictions, v_loss,
                    v_prediction, t_loss, t_prediction = session.run(
                      [ADAM, W, b, loss[0], train_prediction, loss[1],
                      valid_prediction, loss[2], test_prediction],
                      {data[0]: X_batch, labels[0]: Y_batch, data[1]: validData,
                      labels[1]: validTarget, data[2]: testData, labels[2]: testTarget})
            # After each epoch, store the losses and accuracies to plot graphs
            training_loss.append(l)
            validating_loss.append(v_loss)
            testing_loss.append(t_loss)
            train_acc.append(accuracy(predictions, Y_batch))
            valid_acc.append(accuracy(v_prediction, validTarget))
            test_acc.append(accuracy(t_prediction, testTarget))

    plt.figure(1)
    title1 = "Logistic Regresssion with Numpy \n Final Test Loss = %.2f"
              %(testing_loss[len(testing_loss)-1]*100) + "%"
    plt.plot(range(epochs),training_loss)
    plt.plot(range(epochs),validating_loss)
    plt.plot(range(epochs),testing_loss)
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title(title1)
    plt.legend(['Training', 'Validation', 'Testing'], loc='best')
    plt.grid()

    plt.figure(2)
    title2 = "Logistic Regresssion with Numpy \n Final Test Accuracy = %.2f"
              %(test_acc[len(test_acc)-1]*100) + "%"
    plt.plot(range(epochs),train_acc)
    plt.plot(range(epochs),valid_acc)
    plt.plot(range(epochs),test_acc)
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.title(title2)
    plt.legend(['Training', 'Validation', 'Testing'], loc='best')
    plt.grid()
    plt.show()
    return
```

SGD output for minibatch of size 500 over 700 epochs, learning rate = 0.005 and λ = 0:

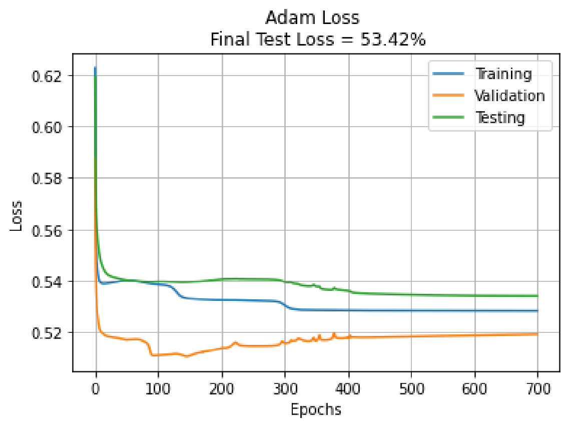


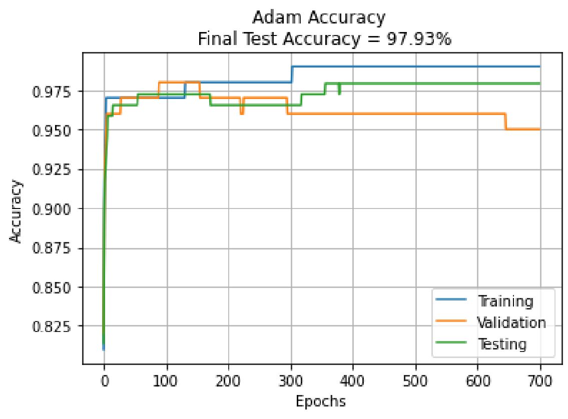


3. Batch Size Investigation

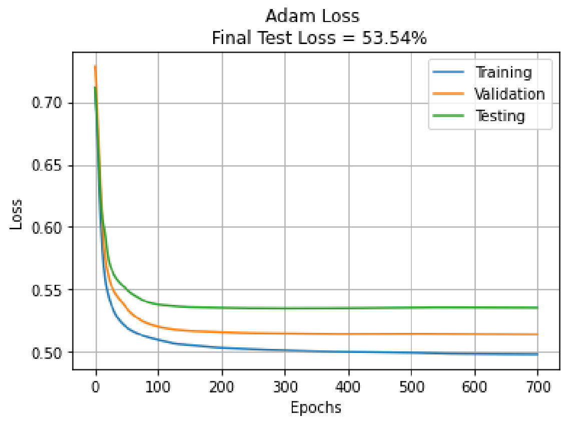
Results from testing my implementation of SGD with 700 epochs and α = 0.001.

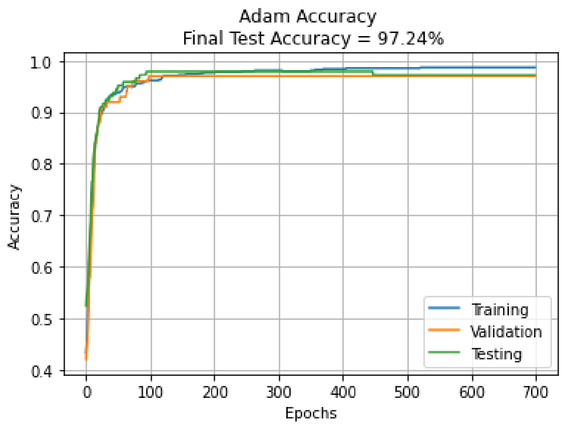
Batch Size = 100



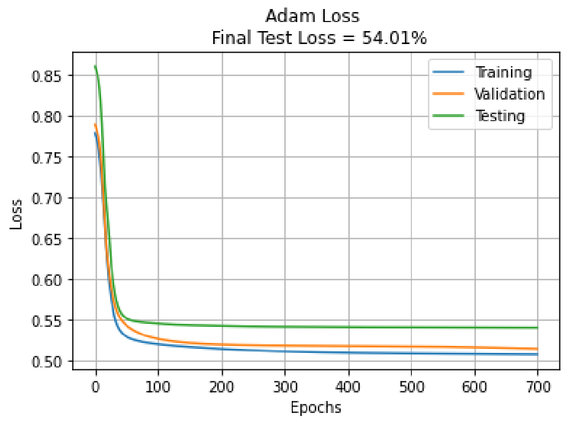


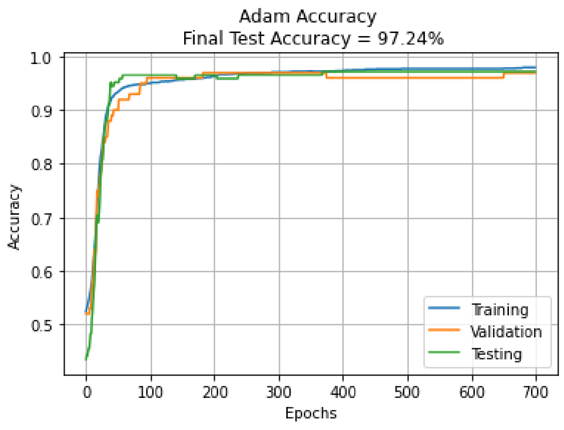
Batch Size = 700





Batch Size = 1750



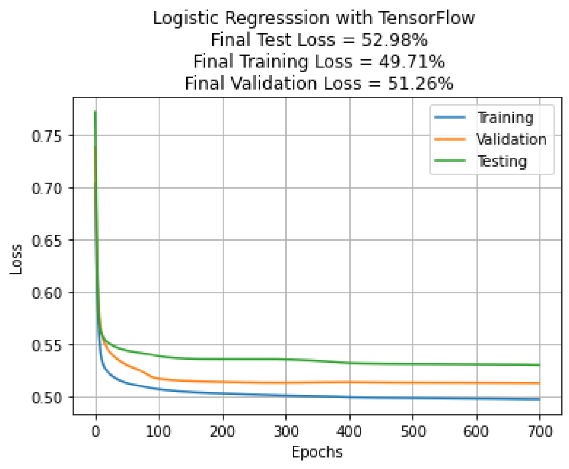


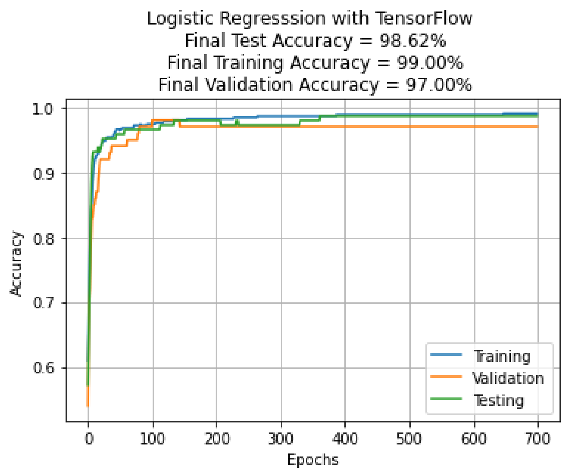
There are not many differences between loss and accuracy curves for batch size 700 and 1750, while for batch size 100 the graph showed a different behaviour that led to the higher final test accuracy (97.93%). As the batch size increases, the trained models become slightly less accurate for the test data set. This could be explained by the fact that mini-batchs have a regularizing effect on the model in the way of adding extra noise to the gradient estimations; this helps led to better generalization. It is also noticeable that, as the batch size increases, the noise becomes smaller and therefore there is a loss on the regularizing effect that causes the lowering of the test accuracy.

4. Hyperparameter Investigation

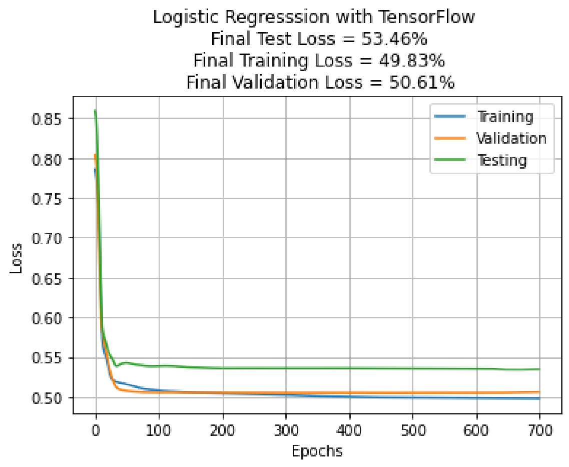
Results from testing my implementation of SGD with 700 epochs, batch size = 500 and $\alpha = 0.001$.

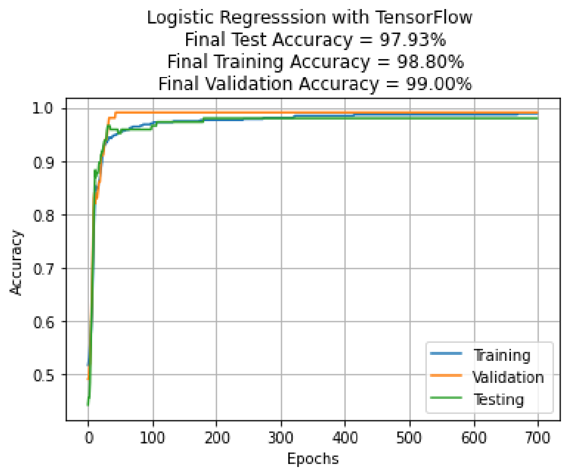
$\beta_1 = 0.95$



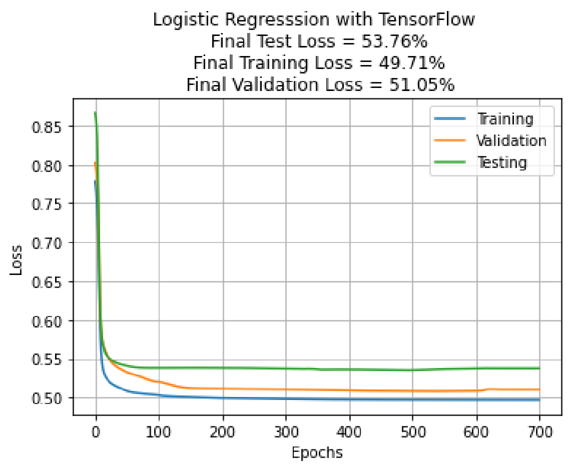


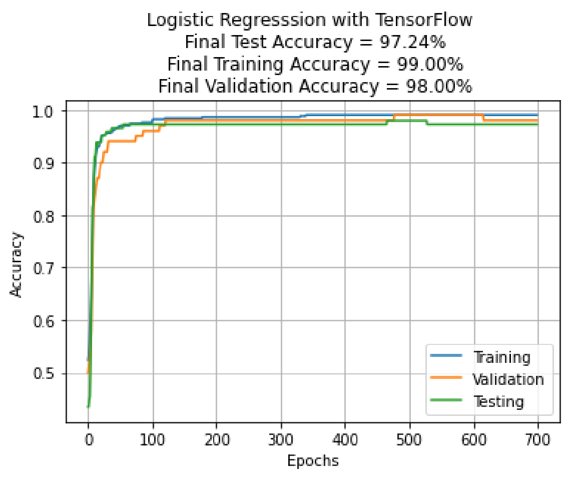
$\beta_1 = 0.99$



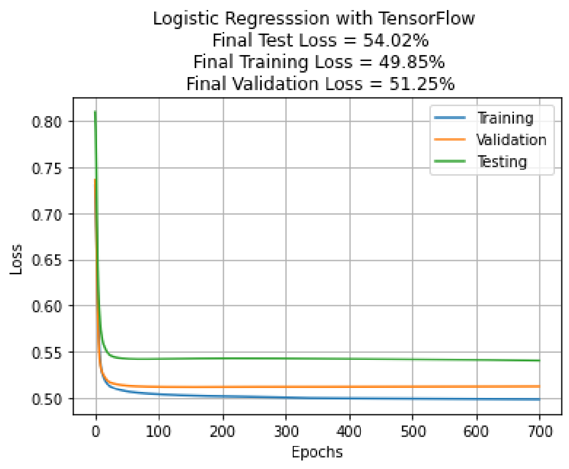


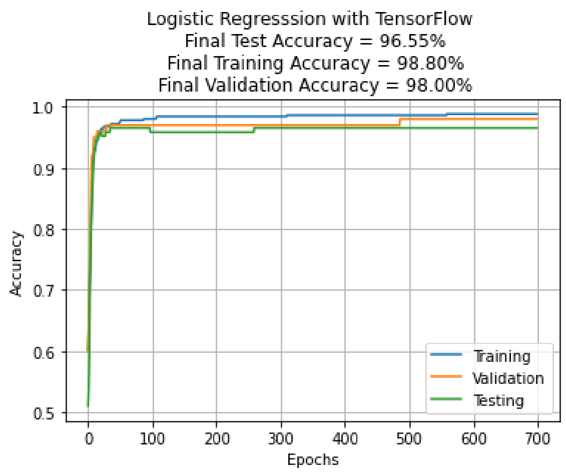
$\beta_2 = 0.99$



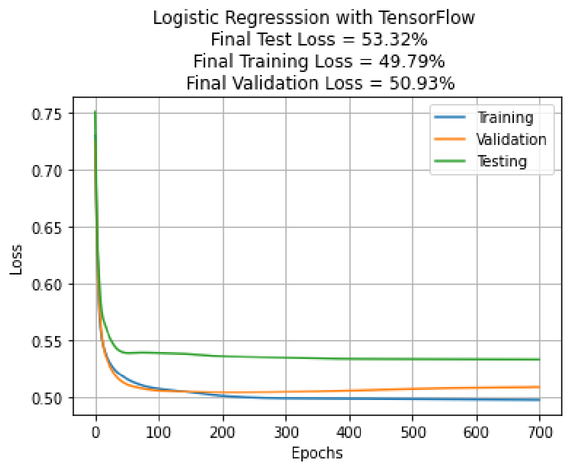


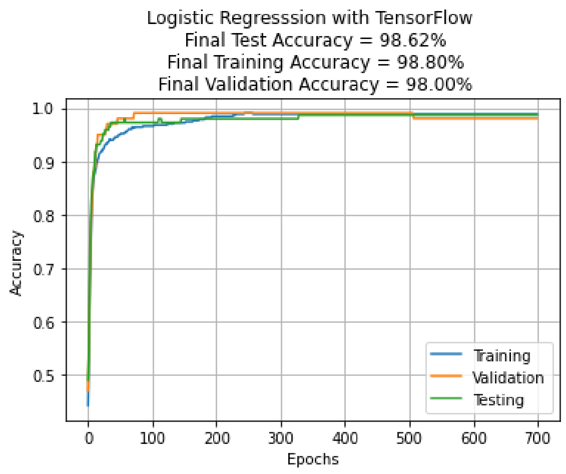
$\beta_2 = 0.9999$



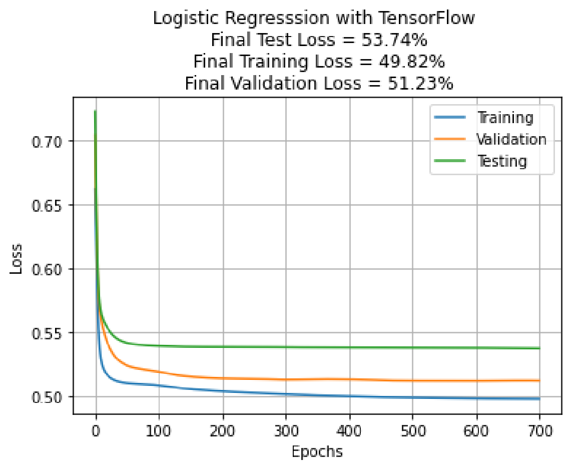


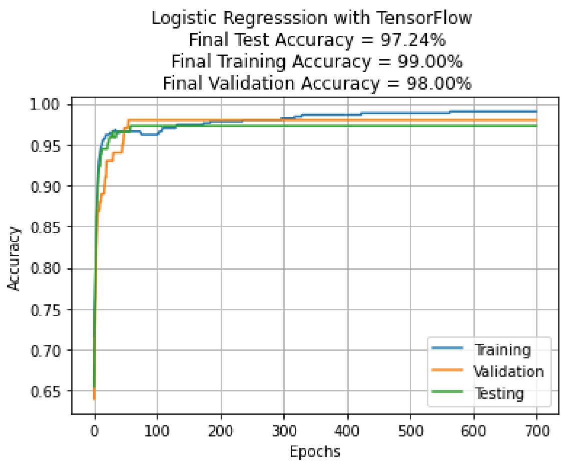
ϵ = 1e-9





ϵ = 1e-4





Hyperparameter Report

Name	B_1=0.95	B_1=0.99	B_2=0.99	B_2=0.9999	Ep=1e-9	Ep=1e-4
Test Loss	52.98%	53.46%	53.76%	54.02%	53.32%	53.74%
Training Loss	49.71%	49.83%	49.71%	49.85%	49.79%	49.82%
Validation Loss	51.26%	50.61%	51.05%	51.25%	50.93%	51.23%
Test Accuracy	98.62%	97.93%	97.24%	96.55%	98.62%	97.24%
Training Accuracy	99.00%	98.80%	99.00%	98.80%	98.80%	99.00%
Validation Accuracy	49.71%	99.00%	98.00%	98.80%	98.00%	98.00%

The hyperparameter β1 and β2 are both used for speeding up the gradient descent. They are able to make the model converges faster than the standard gradient descent algorithm.

β1 is the exponential decay rate for the first moment estimates, which adds momentum into consideration when calculating the gradient descent. An accurate choice of β1 can help to adjust each step size and direction, leading to faster convergent. The plot for β1=0.95 shows fewer oscillations at the beginning of the curves and reaches a higher accuracy (98.62%). Since β1 calculates current weight values by taking historical values into consideration, the higher the β1 the more historical values are taken into consideration to calculate the current weights.

Therefore, when the β1 equals 0.99, the model heavily relies on the historical values, which gives less accuracy (97.93%) and more oscillations.

β2 is the exponential decay rate for the second-moment estimates. It works similar to β1; it can narrow the direction of the oscillation and enlarge the steps in the wider direction. From the data reported in the table, as we can see, the test accuracy is higher after 700 epochs with
β2 = 0.99 (97.24%) and lower when β2=0.9999 (96.55%).

The ϵ values tested are very small, this variable is used to prevent a division by zero during the implementation. The test accuracies for both $\epsilon_{(1e-9)}$and $\epsilon_{(1e-4)}$ are almost the same (98.80% and 99.00% respectively) . The only small difference that can be perceived is the slightly bigger oscillation at the beginning of the curve for ϵ=(1e-4), while the cure for ϵ=(1e-9) tend to be more smooth.

5. Comparison against Batch GD

From the results reported in parts 1 and 2, we are able to confidently state that the overall training model performance using Stochastic Gradient Descent algorithm is much better than using GD.

The performance of the SGD is better in terms of loss, and accuracy and speed and this are due for the following reasons:

1. Adam optimizer leads to a faster convergence rate by using a more accurate step direction towards minimum value.
2. Adam optimizer adds momentum when determining the weights, which helps the model to escape shallow local minimums.
3. Using smaller sized batches allow for a faster calculation time compared to using the whole data set to compute.