ECE421 - Winter 2022
Assignment 1: Logistic Regression

Faranak Dayyani, Student Number: 1002373674

Part 1: Logistic Regression with Numpy

1. Loss Function and Gradient

The total loss function is as follows:

$$\begin{aligned}\mathcal{L} &= \mathcal{L}_{\mathrm{CE}} + \mathcal{L}_{\mathrm{w}} \\ &= \frac{1}{N}\sum_{n=1}^{N}\left[-y^{(n)}\log\hat{y}\left(\mathrm{x}^{(n)}\right) - \left(1-y^{(n)}\right)\log\left(1-\hat{y}\left(\mathrm{x}^{(n)}\right)\right)\right] + \frac{\lambda}{2}\|\mathrm{w}\|_2^2\end{aligned}$$

The Gradient of the loss function with respect to weight (w):

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

The Gradient of the loss function with respect to bias (b):

$$\frac{\partial L}{\partial b} = \hat{y} - y$$

Code snippet for Loss function:

```
# sigmoid function
def sigmoid(x,w):
    z = np.dot(x, w)
    s = 1/(1 + np.exp(-z))

    return s


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

    global LW, L, LCE,LCE_mean, y_hat

    y_hat = sigmoid(x, w)

    w = np.delete(w,-1)
    w = w.reshape(1,784)
    LCE = ((-1*y)*(np.log(y_hat))) - ((1 - y)*np.log(1- y_hat))
    LCE_mean = np.mean(LCE) #cross entropy loss value

    w_norm = np.linalg.norm(w)  #takes the norm L2
    LW = (reg/2)*((w_norm)**2) #regularization loss
    L = LCE_mean + LW   #total loss

    return L
```

Figure 1: Loss Function code snippet

Code snippet for Gradient of the Loss function:

```
def grad_loss(w, b, x, y, reg):

    global Lgrad_wrt_w, Lgrad_wrt_b

    y_hat = sigmoid(x, w)

    m = len(x)
    y_result = y_hat - y
    x_tran = x.T
    Lgrad_wrt_w = ((1/m)*(np.dot(x_tran, y_result))) + (reg*w)
    Lgrad_wrt_b = Lgrad_wrt_w[784,0]
    Lgrad_wrt_w = np.delete(Lgrad_wrt_w,-1)
    Lgrad_wrt_w = Lgrad_wrt_w.reshape(784,1)
    return Lgrad_wrt_w, Lgrad_wrt_b
```

Figure 2: Gradient of the Loss Function code snippet

2. Gradient Descent Implementation

```
def grad_descent(w, b, x, y, xvalid, yvalid, xtest, ytest, alpha, epochs, reg, error_tol):

    global w_updated, b_updated, Ltrain, Lvalid, Larr_train, xaxis, Larr_train_a, Larr_valid, Larr_valid_a, y_hat_train, y_hat_valid
    Larr_train = 0
    Larr_valid = 0
    Aarr_train = 0
    Aarr_valid = 0

    for i in range (epochs):

        # loss data
        Ltrain = loss(w, b, x, y, reg)
        Lvalid = loss(w, b, xvalid, yvalid, reg)
        Ltest = loss(w, b, xtest, ytest, reg)

        # accuracy data
        y_hat_train = sigmoid(x, w)
        y_hat_train[y_hat_train >= 0.5] = 1
        y_hat_train[y_hat_train < 0.5] = 0
        Atrain = np.mean(y_hat_train == y)
        Atrain = Atrain*100

        y_hat_valid = sigmoid(xvalid, w)
        y_hat_valid[y_hat_valid >= 0.5] = 1
        y_hat_valid[y_hat_valid < 0.5] = 0
        Avalid = np.mean(y_hat_valid == yvalid)
        Avalid = Avalid*100

        y_hat_test = sigmoid(xtest, w)
        y_hat_test[y_hat_test >= 0.5] = 1
        y_hat_test[y_hat_test < 0.5] = 0
        Atest = np.mean(y_hat_test == ytest)
        Atest = Atest*100

        Lgrad_wrt_w, Lgrad_wrt_b = grad_loss(w, b, x, y, reg)


        b = w[784,0]
        w = np.delete(w,-1)
        w = w.reshape(784,1)
```

Figure 3: Gradient Descent function code snippet – 1

```python
        w_updated = w - (alpha*Lgrad_wrt_w)
        b_updated = b - (alpha*Lgrad_wrt_b)

        w_difference = w_updated - w
        error = np.linalg.norm(w_difference)

        if error < error_tol:

            print("error is less than error tolerance")

            # ---- plotting Loss -----
            # array for plotting train data
            Larr_train = np.append(Larr_train,Ltrain)
            Larr_train_a = Larr_train[1:epochs+1]

            #array for plotting validation data
            Larr_valid = np.append(Larr_valid,Lvalid)
            Larr_valid_a = Larr_valid[1:epochs+1]

            xaxis = np.linspace(0, epochs, num=epochs)
            plt.plot(xaxis,Larr_train_a)
            plt.plot(xaxis,Larr_valid_a)
            plt.xlabel('Epochs')
            plt.ylabel('Loss')
            plt.title('Loss vs. Number of Epochs - regularization = 0.5')
            plt.legend(['trainData','validData'])
            plt.grid()
            plt.show()
            plt.figure()

            # ---- plotting Accuracy -----
            #array for plotting train data
            Aarr_train = np.append(Aarr_train, Atrain)
            Aarr_train_a = Aarr_train[1:epochs+1]

            #array for plotting validation data
            Aarr_valid = np.append(Aarr_valid,Avalid)
            Aarr_valid_a = Aarr_valid[1:epochs+1]
```
Figure 4: Gradient Descent function code snippet – 2

```
            xaxis = np.linspace(0, epochs, num=epochs)
            plt.plot(xaxis,Aarr_train_a)
            plt.plot(xaxis,Aarr_valid_a)
            plt.xlabel('Epochs')
            plt.ylabel('Accuracy (%)')
            plt.title('Accuracy vs. Number of Epochs - regularization = 0.5')
            plt.legend(['trainData','validData'])
            plt.grid()
            plt.show()

            print("error is less than error_tol")
            break;

        w = w_updated
        b = b_updated
        w = np.append(w,b)
        w = w.reshape(785,1)

        # ---- Loss arrays -----
        # array for plotting train data
        Larr_train = np.append(Larr_train,Ltrain)
        Larr_train_a = Larr_train[1:epochs+1]

        #array for plotting validation data
        Larr_valid = np.append(Larr_valid,Lvalid)
        Larr_valid_a = Larr_valid[1:epochs+1]

        # ---- Accuracy arrays -----
        #array for plotting train data
        Aarr_train = np.append(Aarr_train, Atrain)
        Aarr_train_a = Aarr_train[1:epochs+1]

        #array for plotting validation data
        Aarr_valid = np.append(Aarr_valid,Avalid)
        Aarr_valid_a = Aarr_valid[1:epochs+1]
```

Figure 5: Gradient Descent function code snippet – 3

```
    # ---- plotting Loss -----
    xaxis = np.linspace(0, epochs, num=epochs)
    plt.plot(xaxis,Larr_train_a)
    plt.plot(xaxis,Larr_valid_a)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Loss vs. Number of Epochs - regularization = 0.5')
    plt.legend(['trainData','validData'])
    plt.grid()
    plt.show()

    plt.figure()

    # ---- plotting Accuracy -----
    xaxis = np.linspace(0, epochs, num=epochs)
    plt.plot(xaxis,Aarr_train_a)
    plt.plot(xaxis,Aarr_valid_a)
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.title('Accuracy vs. Number of Epochs - regularization = 0.5')
    plt.legend(['trainData','validData'])
    plt.grid()
    plt.show()

    print("Training Loss:",Ltrain)
    print("Validation Loss:",Lvalid)
    print("Testing Loss:",Ltest)
    print("Training Accuracy:",Atrain)
    print("Validation Accuracy:",Avalid)
    print("Testing Accuracy:",Atest)


    return w_updated, b_updated
```

Figure 6: Gradient Descent function code snippet – 4

3. Tuning the Learning Rate

Table 1: The table below shows the loss and accuracy values for different learning rate (α) values:

Data Type	α = 0.005	α = 0.001	α = 0.0001
Training Loss	0.054409	0.073337	0.158731
Validation Loss	0.060161	0.083721	0.177008
Testing Loss	0.084959	0.086521	0.158022
Training Accuracy	98.31428	97.8	97.39999
Validation Accuracy	98.0	98.0	97.0
Testing Accuracy	97.93103	97.24137	97.24137

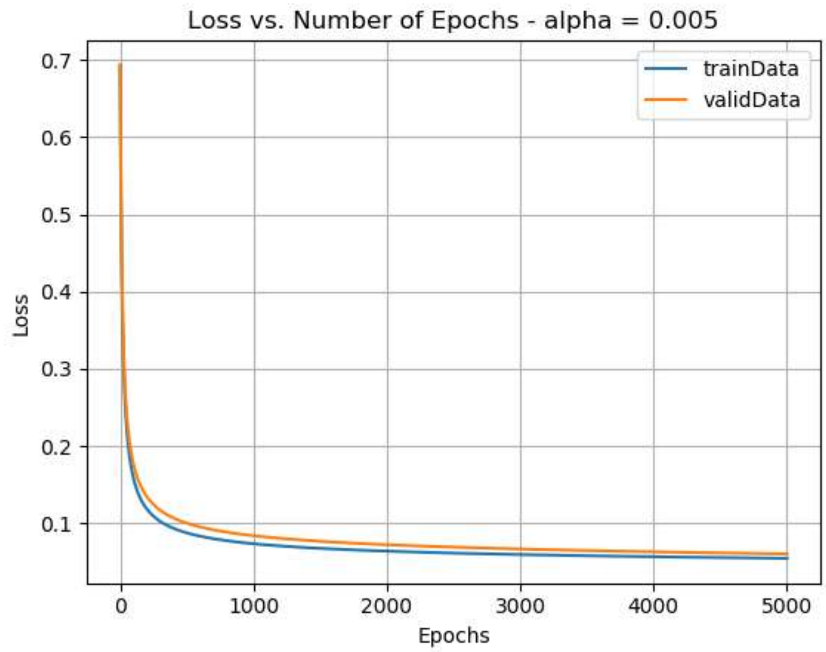

Figure 7: Loss vs. Number of Epochs for α = 0.005

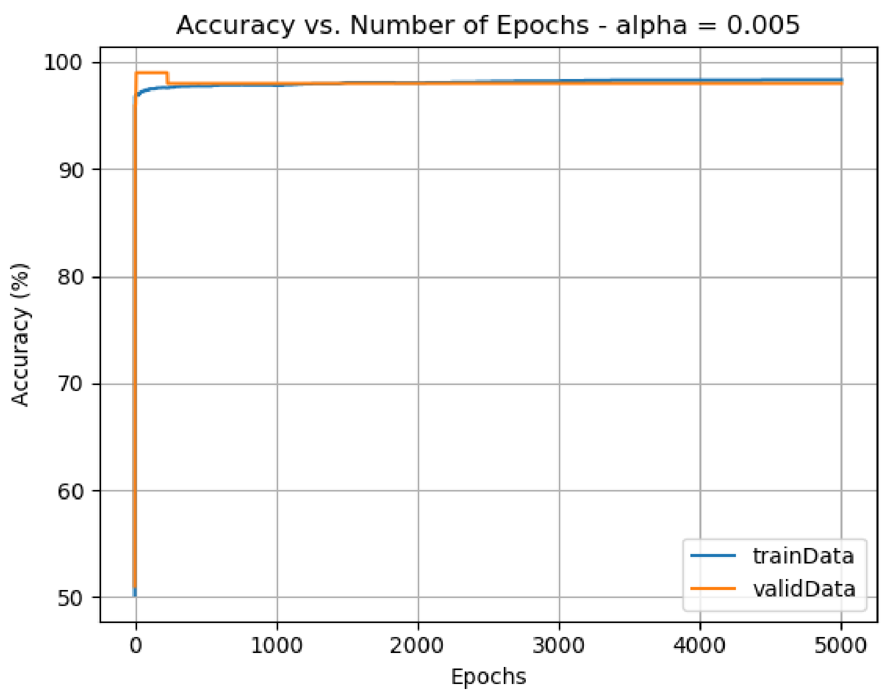

Figure 8: Accuracy vs. Number of Epochs for α = 0.005

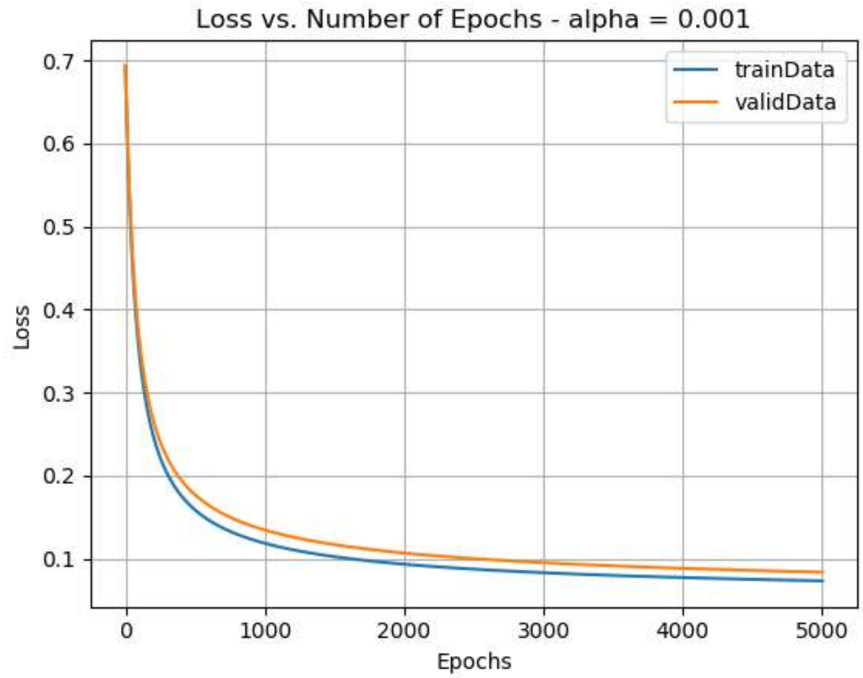

Figure 9: Loss vs. Number of Epochs for α = 0.001

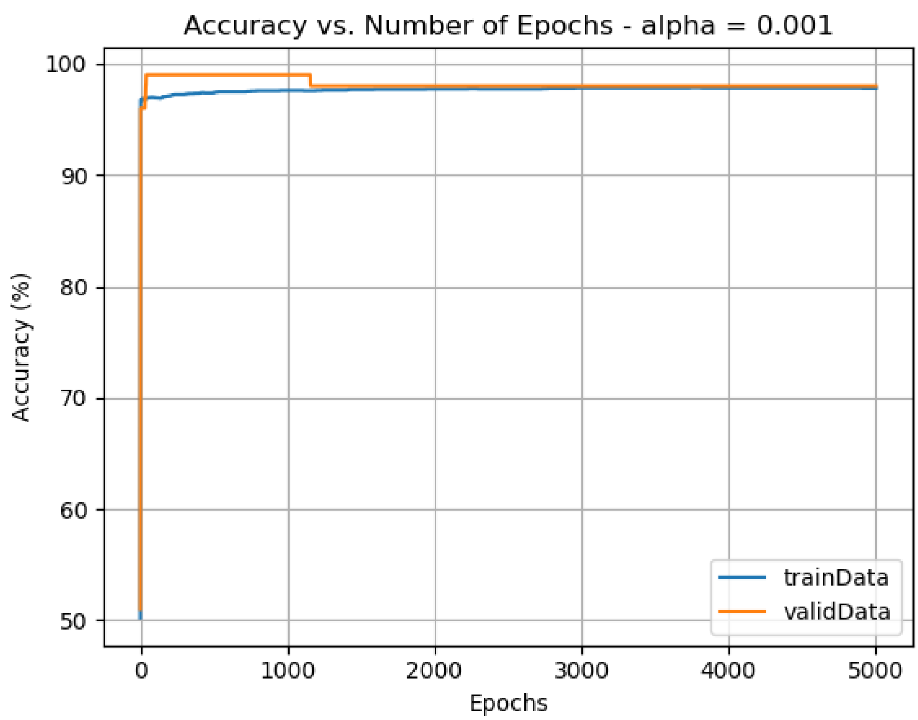

Figure 10: Accuracy vs. Number of Epochs for α = 0.001

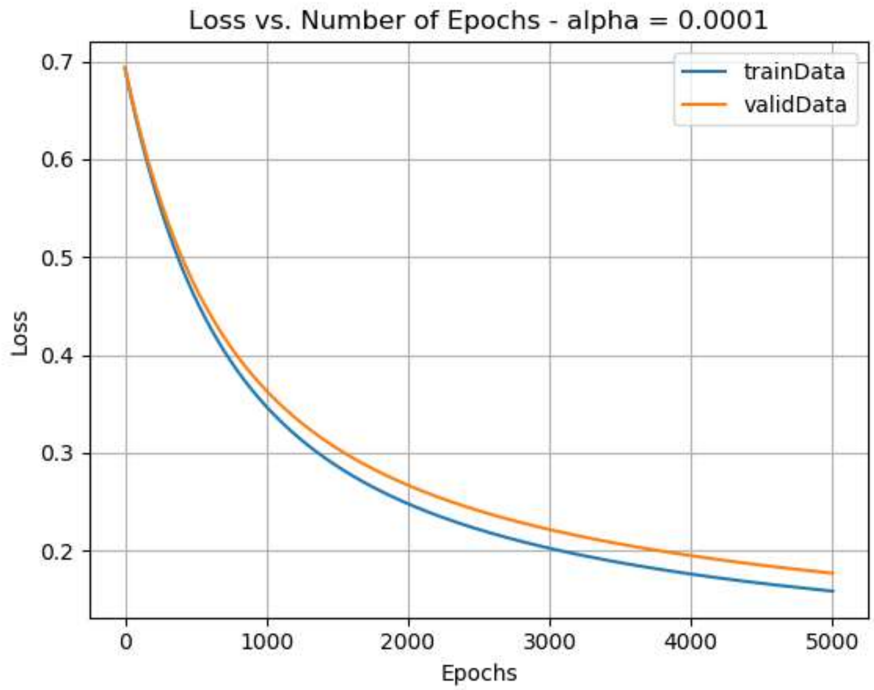

Figure 11: Loss vs. Number of Epochs for α = 0.0001

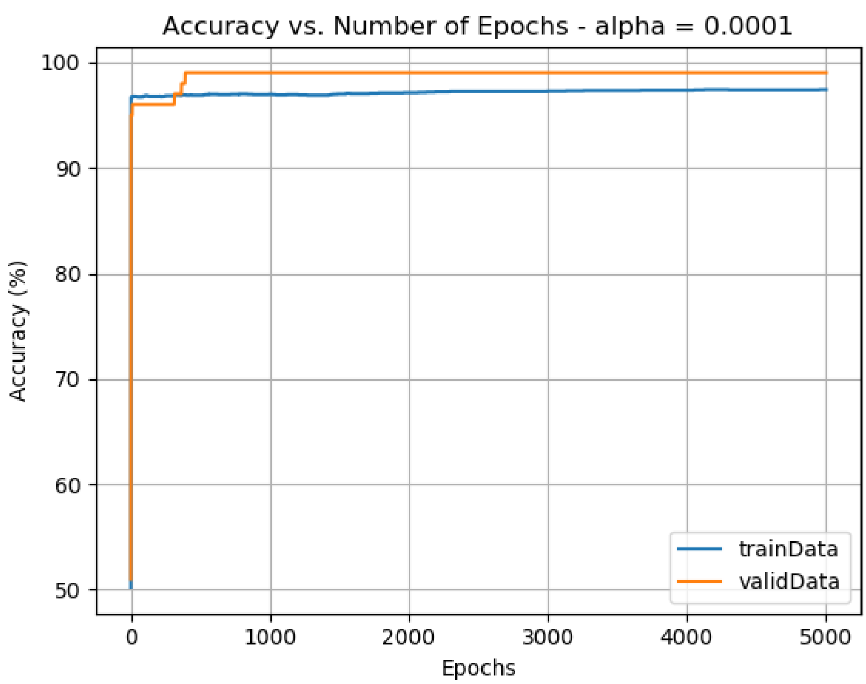

Figure 12: Accuracy vs. Number of Epochs for α = 0.0001

Figure 7-12 above show the loss and accuracy curves for training and validation sets for different learning rate (α) values. Table 1 above displays the loss and accuracy values for training, validation and testing sets for different learning rate (α) values. According to the figures 7-12 and table 1, it can be concluded that the best learning rate is 0.005. This can be supported by the lowest testing loss (0.084959) and highest testing accuracy (97.93103%) observed in table 1. This learning rate also provides the highest accuracy and lowest loss on both of the training and validation sets. The graphs illustrate that the learning rate of 0.005 converges faster than the other two.

In addition, when decreasing the learning rate value to 0.0001, the model has a poor loss compared to the other two values. This is due to the smaller step size which causes it to take more time to reach the global minimum. The smaller the learning rate, the more epochs required to reach the global minimum.

4. Generalization

Table 2: The table below shows the loss and accuracy values for different values of regularization parameter (λ):

Data Type	λ = 0.001	λ = 0.1	λ = 0.5
Training Loss	0.055624	0.114683	0.198496
Validation Loss	0.061497	0.126721	0.216019
Testing Loss	0.085774	0.127311	0.201335
Training Accuracy	98.31429	98.08571	97.74286
Validation Accuracy	98.0	98.0	98.0
Testing Accuracy	97.93103	97.93103	97.93103

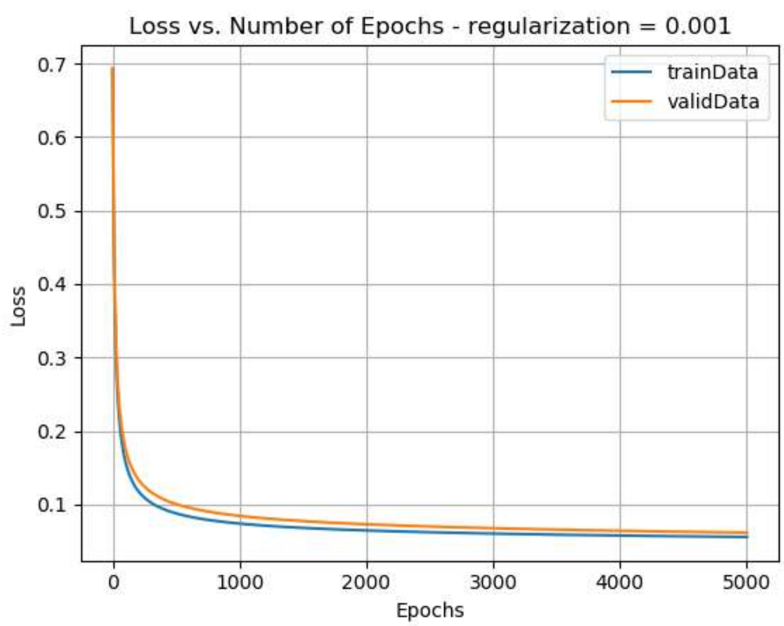

Figure 13: Loss vs. Number of Epochs for λ = 0.001

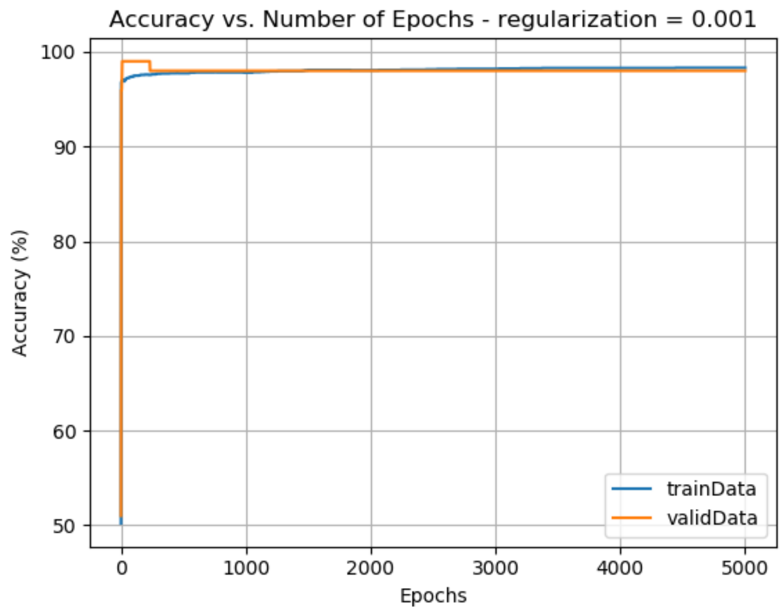

Figure 14: Accuracy vs. Number of Epochs for λ = 0.001

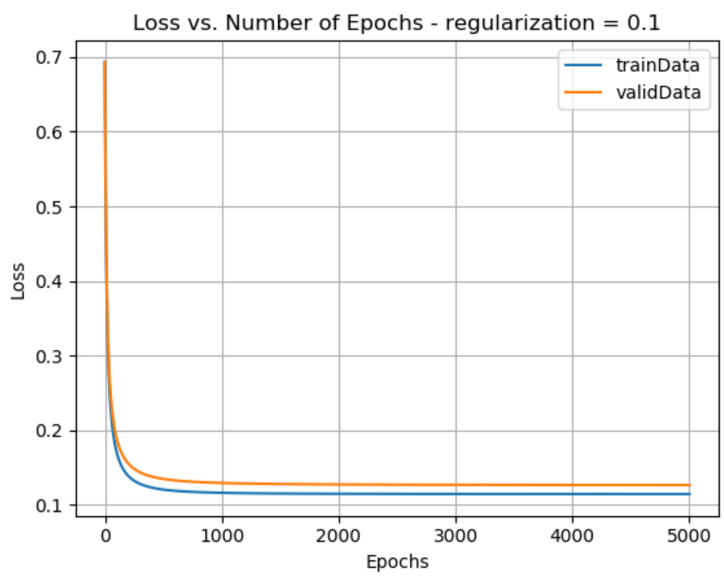

Figure 15: Loss vs. Number of Epochs for λ = 0.1

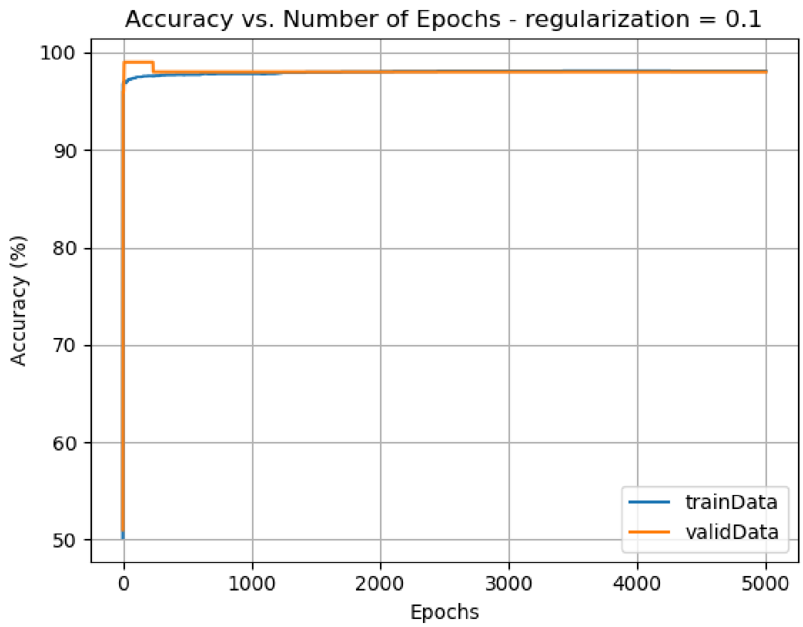

Figure 16: Accuracy vs. Number of Epochs for λ = 0.1

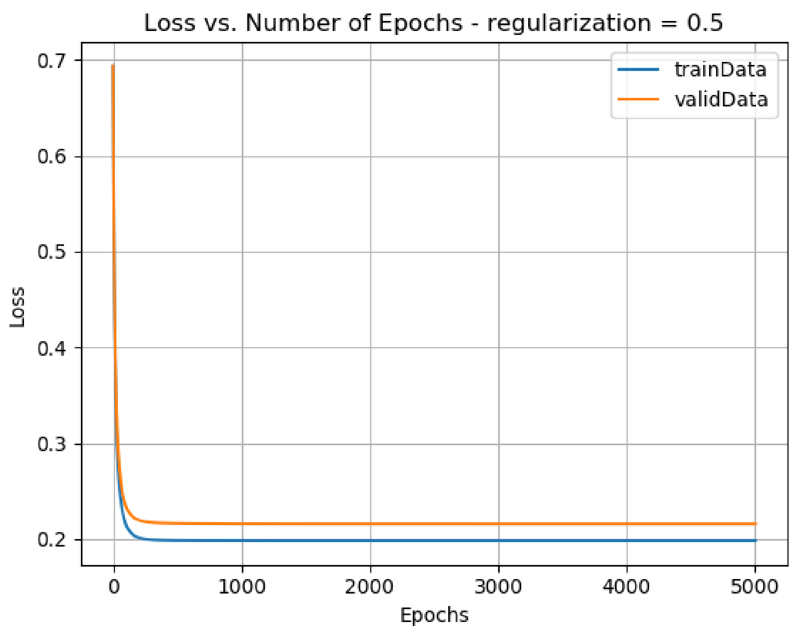

Figure 17: Loss vs. Number of Epochs for λ = 0.5

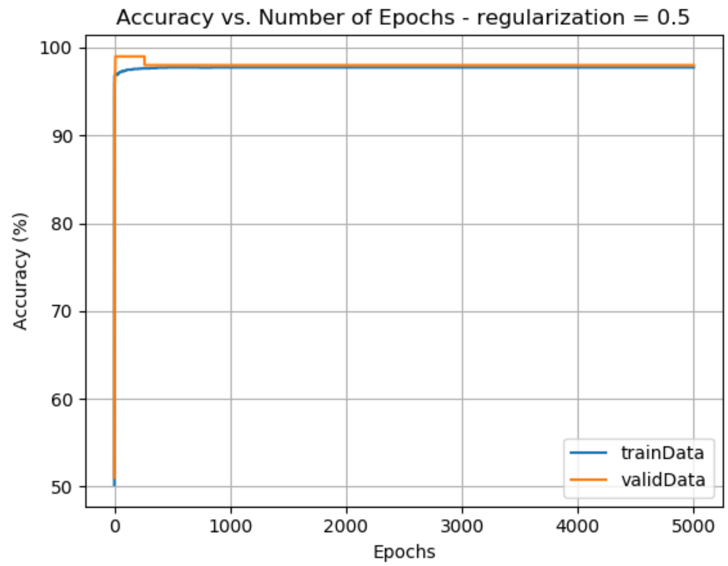

Figure 18: Accuracy vs. Number of Epochs for λ = 0.5

Figure 13-18 above show the loss and accuracy curves for training and validation sets for different values of the regularization (λ) parameter. Table 2 above displays the loss and accuracy values for training, validation and testing sets for different regularization (λ) parameter values. Regularization is used to limit the growth of weight which ensures the curve doesn't fit too well on the training data. According to table 2, it can be seen that as we increase the value of λ, the training loss value also increases. However, lower values of λ contribute less to the loss value in order to prevent overfitting.

According to figures 13-18 and table 2, it can be concluded that λ = 0.5 is the best regularization parameter value as the model avoids overfitting on the training data. The convergence rate is higher for λ = 0.5 compared to the other two values. In addition, the test accuracy (97.93103%) is greater than the training accuracy (97.74286%) for λ = 0.5 which supports the above statement.

Part 2: Logistic Regression in TensorFlow

1. Building the Computational Graph

```
[ ] def buildGraph(xdata, ydata, alpha, ADAM, beta_1=0.9, beta_2=0.999, epsilon=1e-08):
      global loss
      tf.set_random_seed(20)
      xdata_d = xdata.shape[1]

      # --------- Question 1 - Part a -------------------
      #inializing weight and bias tensors
      w = tf.truncated_normal_initializer(mean=0, stddev=0.5)
      w = tf.get_variable("w", (xdata_d, 1), initializer=w)
      b = tf.constant_initializer(0.0)
      b = tf.get_variable("b", (1, ), initializer=b)

      # --------- Question 1 - Part b -------------------
      # placeholders for data, label and reg
      x = tf.placeholder(tf.float32, shape=(None, xdata_d))
      y = tf.placeholder(tf.float32, shape=(None, 1))
      reg = tf.placeholder(tf.float32)

      # --------- Question 1 - Part c -------------------
      # Loss function
      z = tf.add(tf.matmul(x, w), b)
      y_hat = tf.nn.sigmoid(z, name="y_hat")
      LCE = tf.nn.sigmoid_cross_entropy_with_logits(labels=y, logits=z)
      LCE = tf.reduce_mean(LCE) #cross entropy loss
      LW = (reg/2) * tf.reduce_sum(tf.square(w)) #regularizaton term loss
      loss = tf.add(LCE, LW) #total loss

      # --------- Question 1 - Part d -------------------
      # optimizer
      if ADAM==True:
        optimizer = tf.train.AdamOptimizer(alpha, beta_1, beta_2, epsilon).minimize(loss)
      else:
        optimizer = tf.train.GradientDescentOptimizer(alpha).minimize(loss)

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

Figure 19: Code snippet of buildGraph function

2. Implementing Stochastic Gradient Descent

```
def SGD(batch_size, epochs, alpha, reg_2, ADAM, beta_1=0.9, beta_2=0.999, epsilon=1e-08):
  global trainData, validData, trainTarget, validTarget, testData, testTarget, loss_2, loss

  N = trainData.shape[0]
  total_batches = int(N/batch_size)
  iterations = total_batches * epochs

  x, w, b, y_hat, y, loss, optimizer, reg = buildGraph(trainData,trainTarget, alpha, ADAM, beta_1, beta_2, epsilon)

  global_init = tf.global_variables_initializer()
  L_train = []
  L_valid = []
  A_train = []
  A_valid = []
  L_test = []
  A_test = []

  with tf.Session() as sess:
    sess.run(global_init)
    for i in range(iterations):
      if (i+1)%total_batches == 0:
        shuffle_index = np.random.choice(N, N, replace=False)
        trainData, trainTarget = trainData[shuffle_index], trainTarget[shuffle_index]
      trainBatch, ybatch = trainData[:batch_size], trainTarget[:batch_size]

      _, w_2, b_2, loss_2 = sess.run([optimizer, w, b, loss], feed_dict={x:trainBatch, y:ybatch, reg:reg_2})
      trainData = np.roll(trainData, batch_size, axis=0)
      trainTarget = np.roll(trainTarget, batch_size, axis=0)
```

Figure 20: Code snippet of SGD function – 1

```
      if (i+1)%total_batches == 0:
        L_train_2, L_valid_2, A_train_2, A_valid_2, L_test_2, A_test_2 = AccValue(w_2, b_2, trainData, trainTarget, reg_2)
        L_train.append(L_train_2)
        L_valid.append(L_valid_2)
        A_train.append(A_train_2)
        A_valid.append(A_valid_2)
        L_test.append(L_test_2)
        A_test.append(A_test_2)

    w_updated, b_updated = sess.run([w, b])

  return w_updated, b_updated, L_train, L_valid, A_train, A_valid, L_test, A_test
```

Figure 21: Code snippet of SGD function – 2

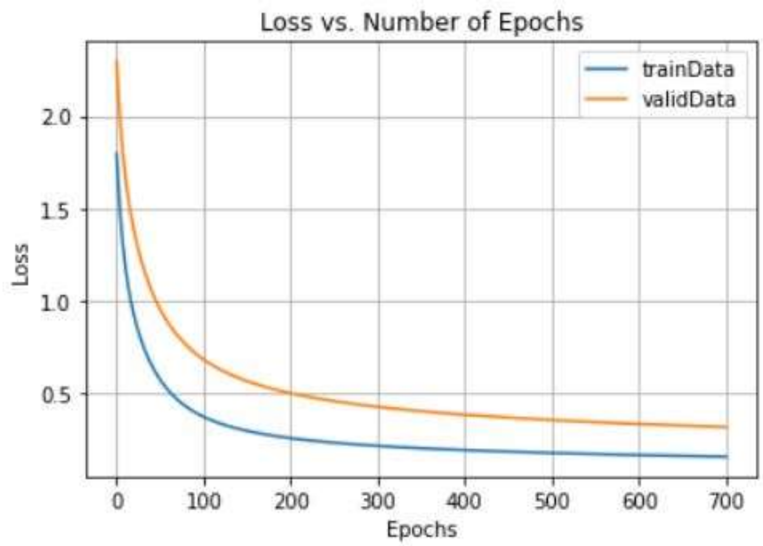

Figure 22: Loss vs. Number of Epochs for SGD

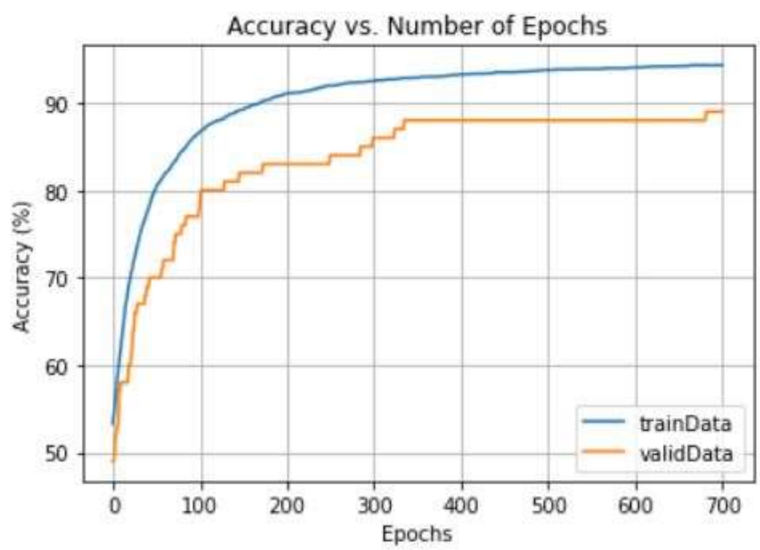

Figure 23: Accuracy vs. Number of Epochs for SGD

3. Batch Size Investigation

Table 3: The table below shows the loss and accuracy values for different batch sizes:

Data Type	Batch size = 100	Batch size = 700	Batch size = 1750
Training Loss	0.005035	0.026014	0.051225
Validation Loss	0.136783	0.110864	0.144830
Testing Loss	0.202145	0.123003	0.138895
Training Accuracy	99.97143	99.02857	97.94286
Validation Accuracy	97.0	96.0	95.0
Testing Accuracy	97.24138	98.62069	97.24138

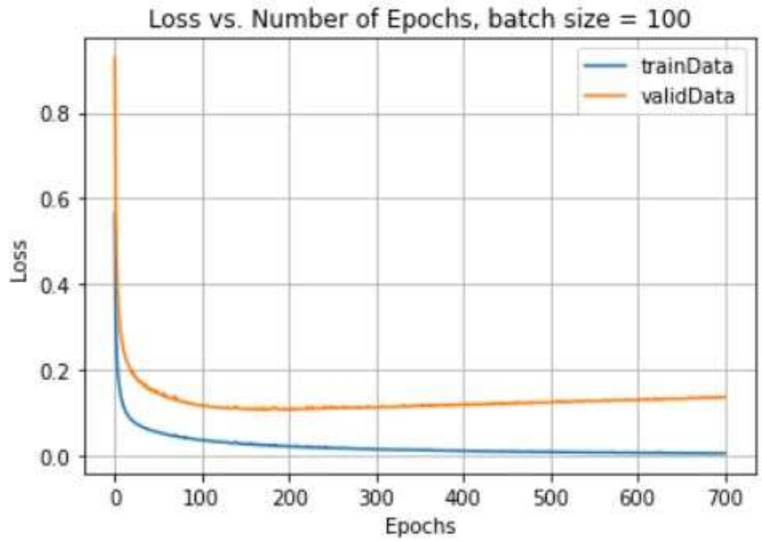

Figure 24: Loss vs. Number of Epochs for SGD – batch size = 100

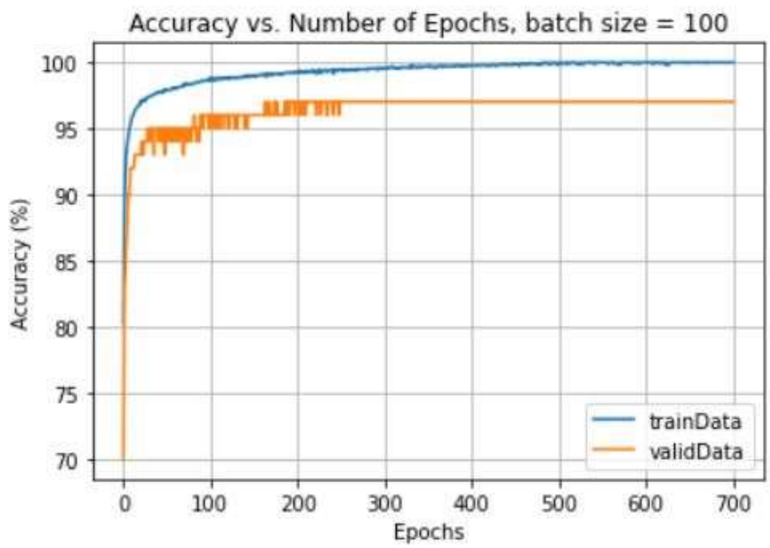

Figure 25: Accuracy vs. Number of Epochs for SGD – batch size = 100

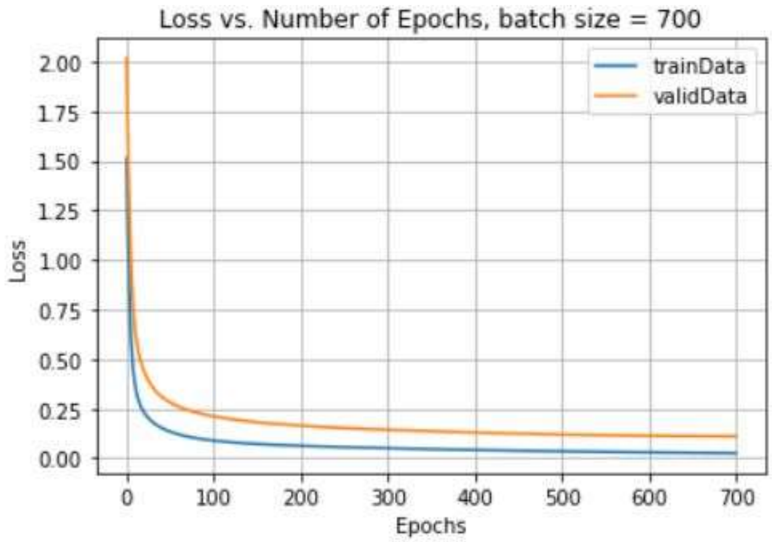

Figure 26: Loss vs. Number of Epochs for SGD – batch size = 700

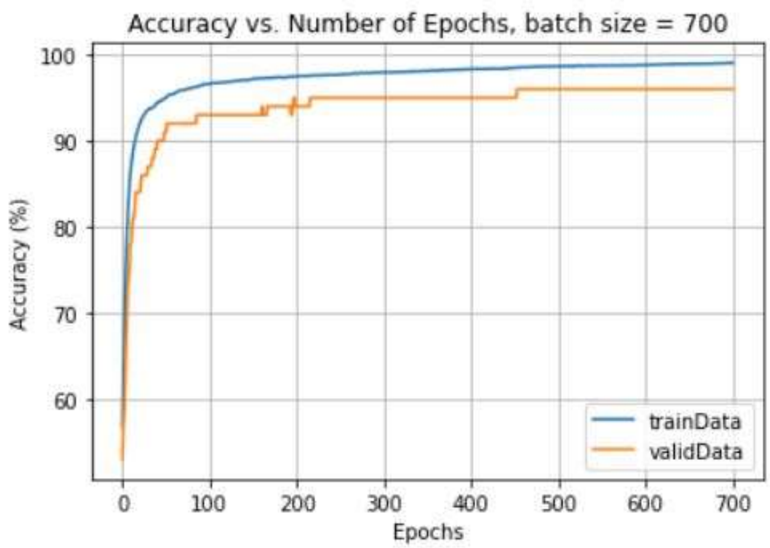

Figure 27: Accuracy vs. Number of Epochs for SGD – batch size = 700

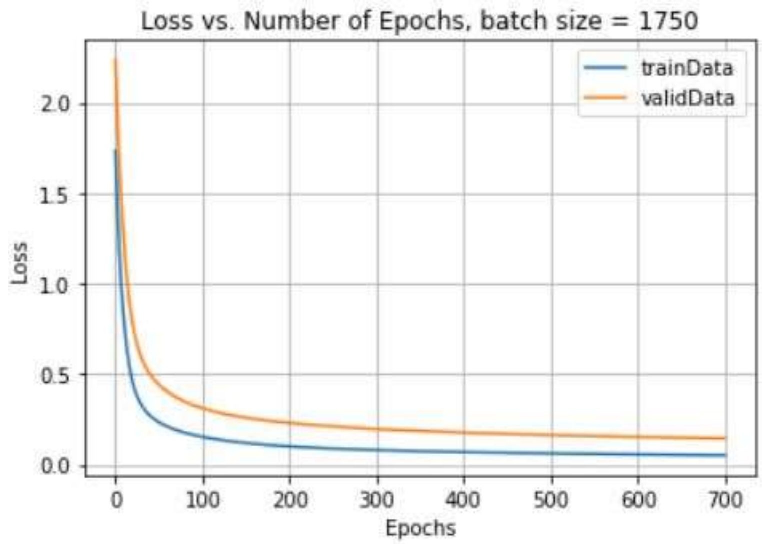

Figure 28: Loss vs. Number of Epochs for SGD – batch size = 1750

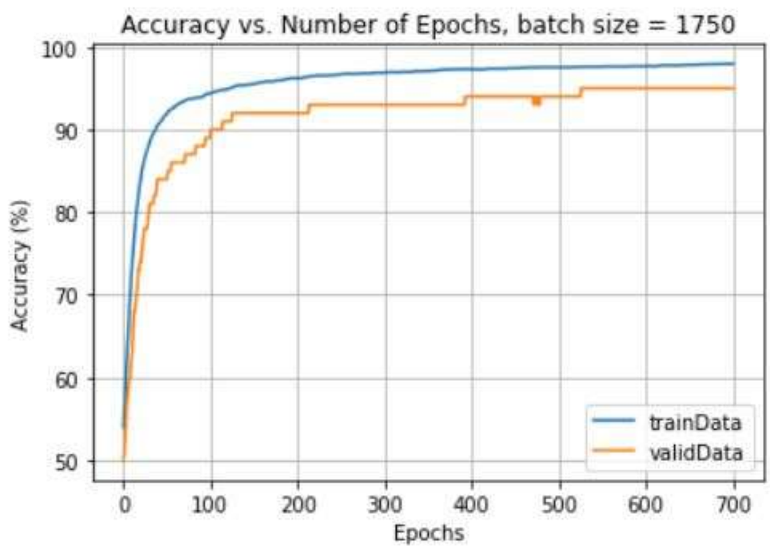

Figure 29: Accuracy vs. Number of Epochs for SGD – batch size = 1750

Figure 24-29 above show the loss and accuracy curves for training and validation sets for different batch sizes. Table 3 above displays the loss and accuracy values for training, validation and testing sets for different batch size values. According to the figures, it can be stated that the convergence rate of the training data for batch size = 100 is slower in comparison with the higher batch sizes. However, the convergence rate for the validation data of batch size = 100 does not reach the global minimum. As seen in figure 27 and 29, the accuracy curves for higher batch sizes are fairly stable. The best batch size value is 100 since it has the highest validation accuracy (97.0%). The highest testing accuracy is for batch size = 700 but when looking at training and validation accuracy values coupled with the testing accuracy, the model performs best with batch size = 100. The small batches have a regularizing effect by adding noise to gradient estimations. This leads to better generalization by helping the model to come out of the local minima. To conclude, as the batch size increases, the effect of regularization decreases which leads to lower test accuracy.

4. Hyperparameter Investigation

Table 4: The table below shows the loss and accuracy values for different Adam hyperparameters:

Data Type	$\beta_1 = 0.95$	$\beta_1 = 0.99$	$\beta_2 = 0.99$	$\beta_2 = 0.9999$	$\varepsilon = 1e - 09$	$\varepsilon = 1e - 4$
Training Loss	0.019307	0.019437	0.011871	0.026001	0.019329	0.019729
Validation Loss	0.109335	0.110801	0.116254	0.1121769	0.108994	0.108917
Testing Loss	0.129573	0.129362	0.146202	0.1229372	0.129105	0.128440
Training Accuracy	99.34574	99.34286	99.6	99.05714	99.34286	99.34286
Validation Accuracy	97.0	97.0	97.0	96.0	97.0	97.0
Testing Accuracy	98.62189	98.62069	97.24138	96.62069	98.62179	98.62069

- Note: the default hyperparameters that were used for Adam are:
    - $\beta_1 = 0.9$
    - $\beta_2 = 0.999$
    - $\varepsilon = 1e - 08$

Table 4 above displays the loss and accuracy values for training, validation and testing sets for different values of Adam hyperparameters. β_1 and β_2 are used for speeding up the gradient descent in Adam.

a) β_1 is the exponential decay rate for the first moment estimate. As seen in table 4 above, $\beta_1 = 0.95$ has higher accuracy values compared to $\beta_1 = 0.99$. The lower the value of β_1, the more importance is given to the historical gradient values for updating the first moment estimate which explains the reason behind $\beta_1 = 0.95$ resulting in higher accuracy values compared to $\beta_1 = 0.99$. To conclude, $\beta_1 = 0.95$ is a better pick due to its higher test accuracy value (98.62189%).
b) β_2 is the exponential decay rate for the second moment estimate. As seen in table 4 above, $\beta_2 = 0.99$ has higher accuracy values compared to $\beta_2 = 0.9999$. The higher the value of β_2, the more weight is given to the square of historical gradient for updating the second moment estimate which explain the reason behind $\beta_2 = 0.99$ resulting in higher accuracy values compared to $\beta_2 = 0.9999$. To conclude, $\beta_2 = 0.99$ is a more ideal pick due to its higher test accuracy value (97.24138%).
c) ε value is used to prevent a division by zero in the implementation of the model. As seen in table 4 above, $\varepsilon = 1e - 09$ has slightly higher accuracy values compared to $\varepsilon = 1e - 4$ while being very close. To conclude, $\varepsilon = 1e - 09$ is a more ideal pick due to its slightly higher test accuracy value (98.62179%).

5. Comparison against Batch GD

Table 5: The table below shows the loss and accuracy values for Stochastic Gradient Descent with Adam and Batch Gradient Descent:

Data Type	ADAM	Batch Gradient Descent
Training Loss	0.012142	0.073337
Validation Loss	0.102934	0.083722
Testing Loss	0.157186	0.0865216
Training Accuracy	99.62857	97.8
Validation Accuracy	97.0	98.0
Testing Accuracy	97.93103	97.24138

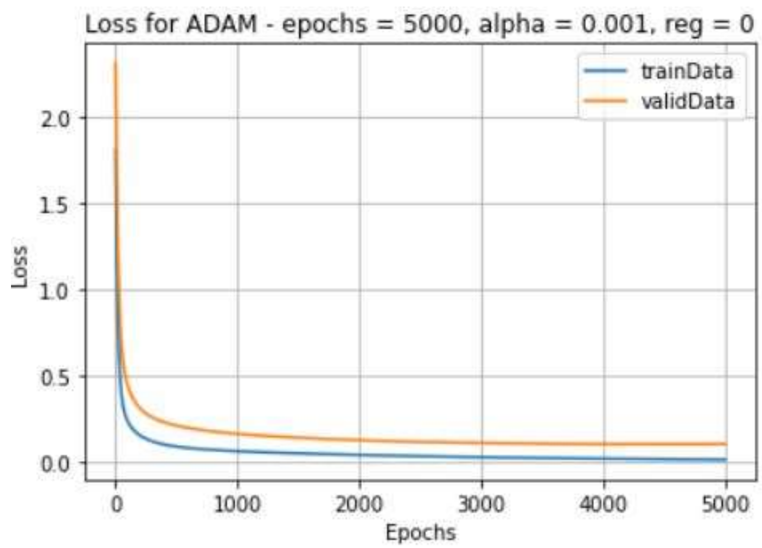

Figure 30: Loss vs. Number of Epochs for SGD with Adam

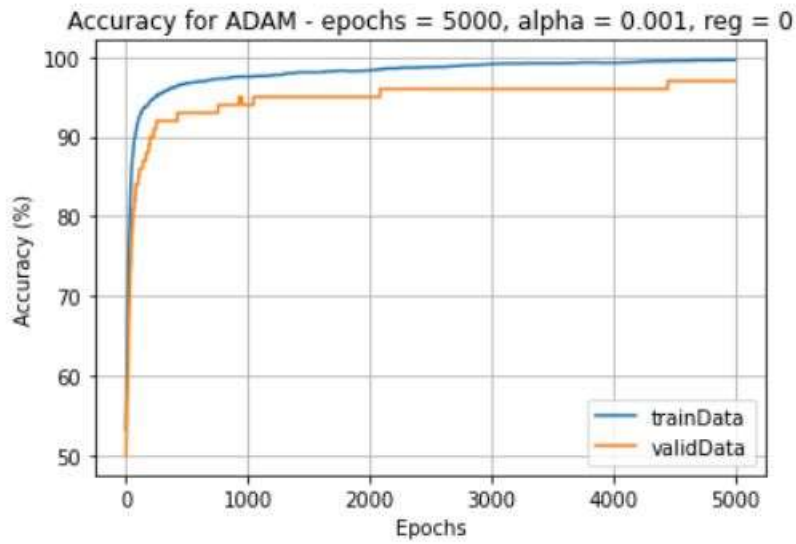

Figure 31: Accuracy vs. Number of Epochs for SGD with Adam

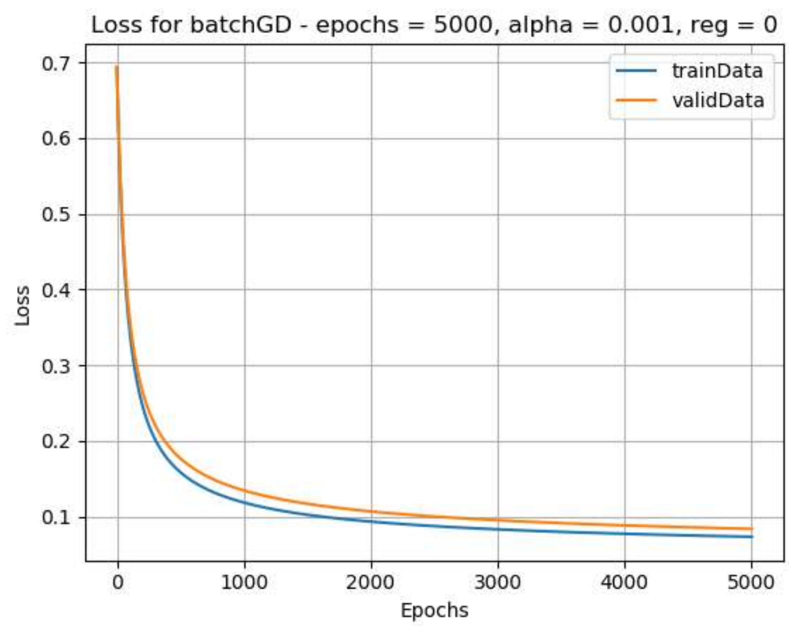

Figure 32: Loss vs. Number of Epochs for Batch Gradient Descent

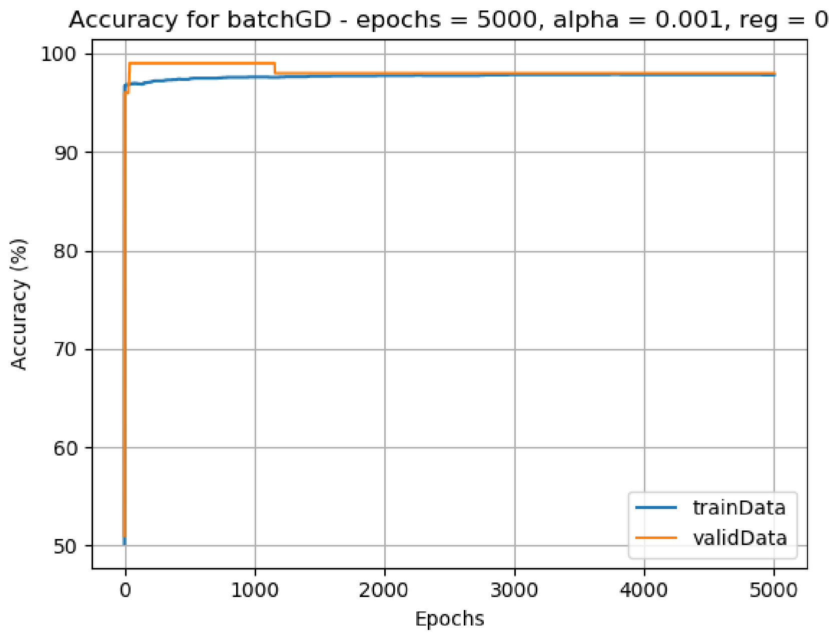

Figure 33: Accuracy vs. Number of Epochs for Batch Gradient Descent

Figures 30-33 above shows the loss and accuracy curves for training and validation sets for Stochastic Gradient Descent (SGD) algorithm with Adam and Batch Gradient Descent (GD) algorithm. Table 5 above illustrates the loss and accuracy values for SGD with Adam and Batch GD with the following hyperparameters:

- Epochs = 5000
- $\lambda = 0$
- $\alpha = 0.001$

The hyperparameters were kept the same for both in order to have a fair assessment of the overall performance of both algorithms. The overall performance of SGD algorithm is better than the Batch GD algorithm due to its use of mini-batches.

According to figures 30 and 32 above, it can be seen that the final loss for SGD is lower than that of Batch GD. Furthermore, the rate of convergence is higher in SGD.

According to figures 31 and 33 above, it can be seen that the final accuracy of SGD is higher than the Batch GD.

To conclude, SGD algorithm with Adam has a better performance in comparison with Batch GD. One of the reasons for this can include that the Adam optimizer aids in a faster convergence rate.