ECE421 - Winter 2022 Assignment 1: Logistic Regression

Mingyu Zheng 1003797661

Due date: February 7

1 Logistic Regression with Numpy

1.1 Loss Function and Gradient

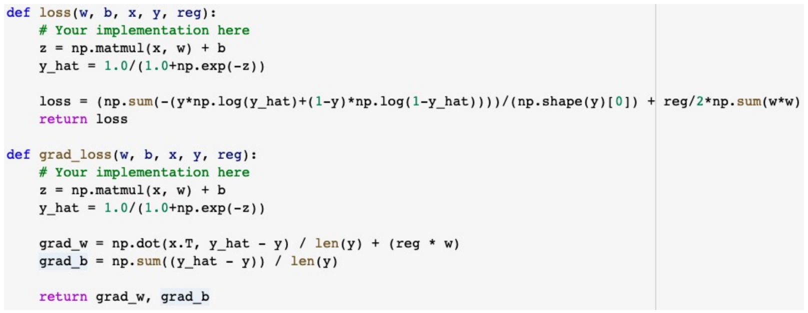
The total loss function is the sum of the cross-entropy loss and the regularization term:

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

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

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

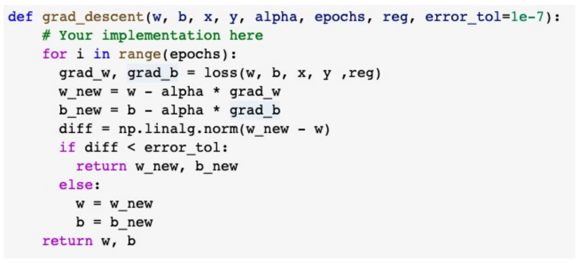
$$\frac{\partial\mathcal{L}}{\partial b} = \frac{1}{N}\sum_{n=1}^{N}(\hat{y}-y) \tag{2}$$



```python
def loss(w, b, x, y, reg):
    # Your implementation here
    z = np.matmul(x, w) + b
    y_hat = 1.0/(1.0+np.exp(-z))

    loss = (np.sum(-(y*np.log(y_hat)+(1-y)*np.log(1-y_hat))))/(np.shape(y)[0]) + reg/2*np.sum(w*w)
    return loss

def grad_loss(w, b, x, y, reg):
    # Your implementation here
    z = np.matmul(x, w) + b
    y_hat = 1.0/(1.0+np.exp(-z))

    grad_w = np.dot(x.T, y_hat - y) / len(y) + (reg * w)
    grad_b = np.sum((y_hat - y)) / len(y)

    return grad_w, grad_b
```

Figure 1: Screenshot of Loss Function

1.2 Gradient Descent Implementation



```
def grad_descent(w, b, x, y, alpha, epochs, reg, error_tol=1e-7):
    # Your implementation here
    for i in range(epochs):
      grad_w, grad_b = loss(w, b, x, y ,reg)
      w_new = w - alpha * grad_w
      b_new = b - alpha * grad_b
      diff = np.linalg.norm(w_new - w)
      if diff < error_tol:
        return w_new, b_new
      else:
        w = w_new
        b = b_new
    return w, b
```

Figure 2: Screenshot of Gradient Descent Function

1.3 Tuning the Learning Rate

```
Training data accuracy:  0.9634285714285714
Valid data accuracy:  0.96
Test data accuracy:  0.9724137931034482
```

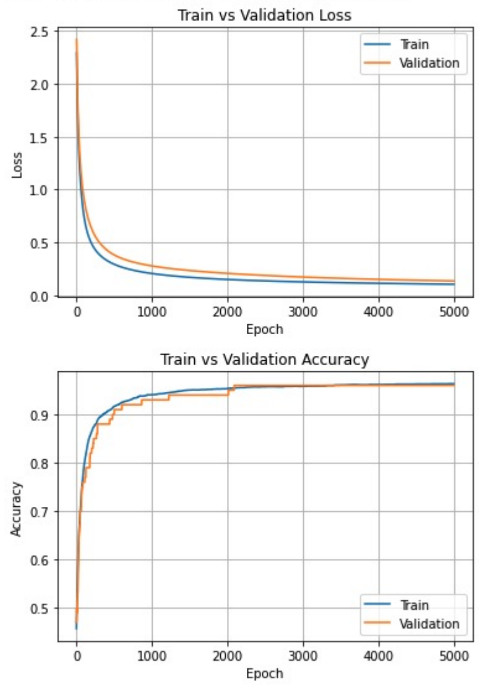


Figure 3: Training and Validation Loss and Accuracy (b=0, epochs=5000, alpha lr=0.005)

```
Training data accuracy:  0.9405714285714286
Valid data accuracy:  0.93
Test data accuracy:  0.9517241379310345
```

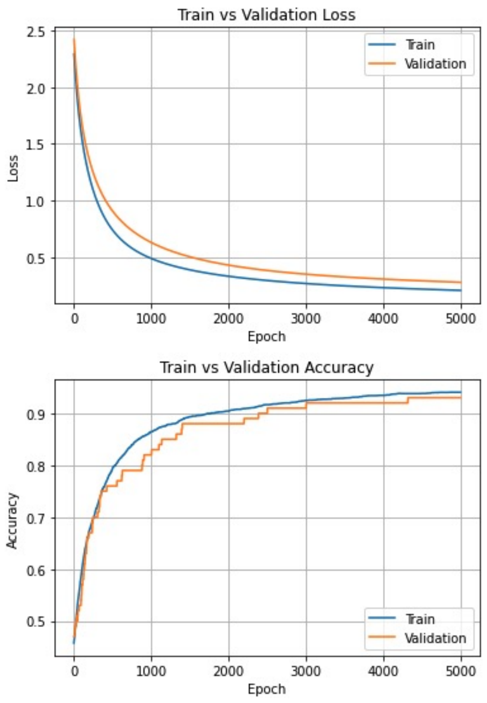
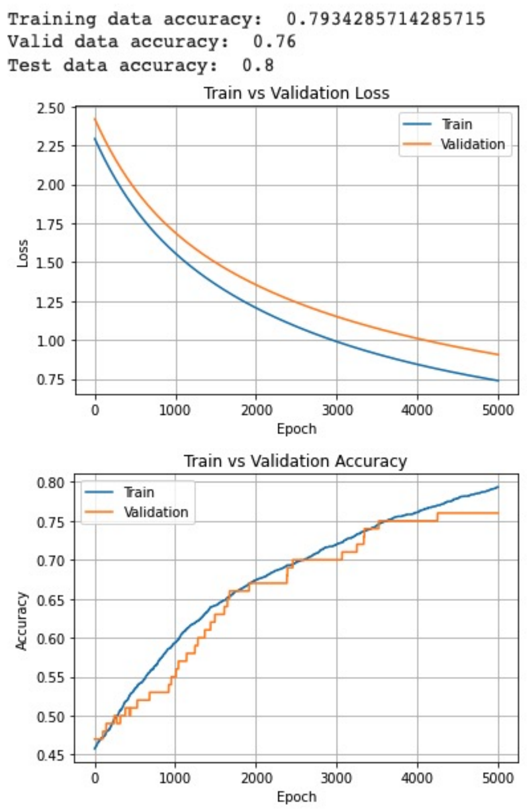


Figure 4: Training and Validation Loss and Accuracy (b=0, epochs=5000, alpha lr=0.001)

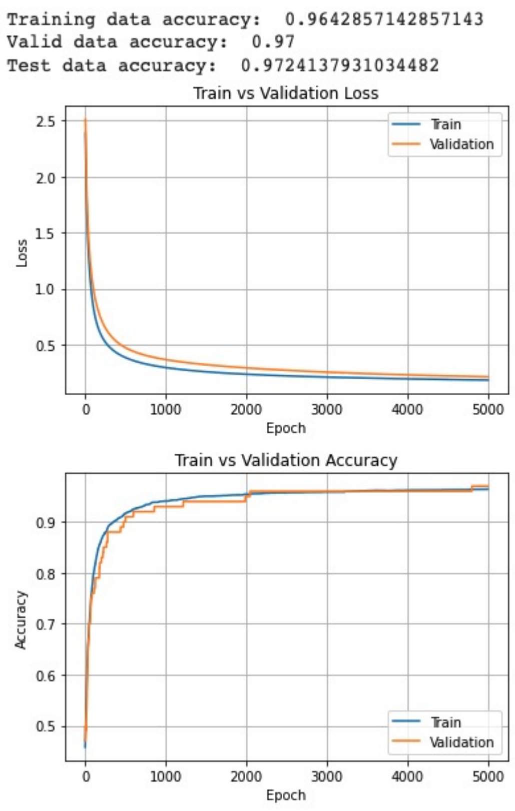


```
Training data accuracy:  0.7934285714285715
Valid data accuracy:  0.76
Test data accuracy:  0.8
```

Figure 5: Training and Validation Loss and Accuracy (b=0, epochs=5000, alpha lr=0.0001)

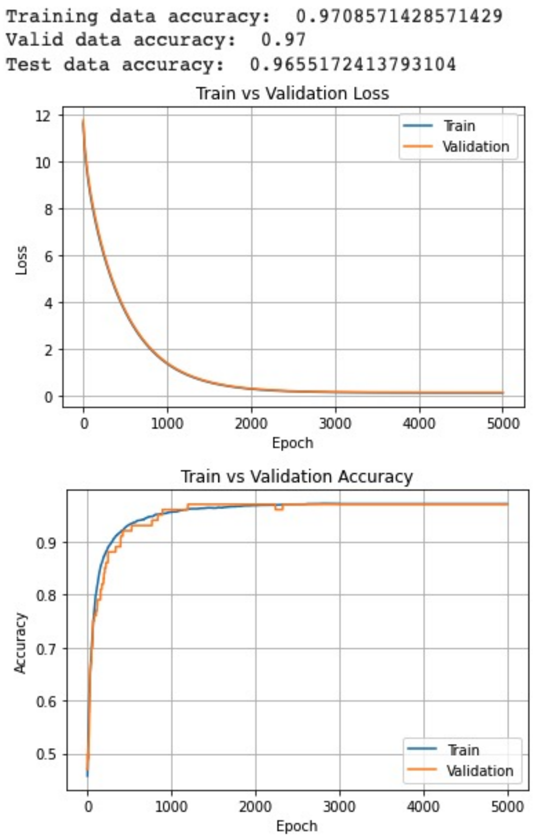
Explain: From the above 3 screenshots, we can tell that the learning rate of 0.005 results in best training and validation accuracy and lowest loss. So 0.005 is the best learning rate compared to 0.001 and 0.0001. With the best learning rate, we got training accuracy of 0.963 and validation accuracy of 0.972.

1.4 Generalization



```
Training data accuracy:  0.9642857142857143
Valid data accuracy:  0.97
Test data accuracy:  0.9724137931034482
```

Figure 6: Training and Validation Loss and Accuracy (b=0, epochs=5000, alpha lr=0.005, reg=0.01)



```
Training data accuracy:  0.9708571428571429
Valid data accuracy:  0.97
Test data accuracy:  0.9655172413793104
```

Figure 7: Training and Validation Loss and Accuracy (b=0, epochs=5000, alpha lr=0.001, reg=0.1)

```
Training data accuracy:  0.9554285714285714
Valid data accuracy:  0.96
Test data accuracy:  0.9517241379310345
```

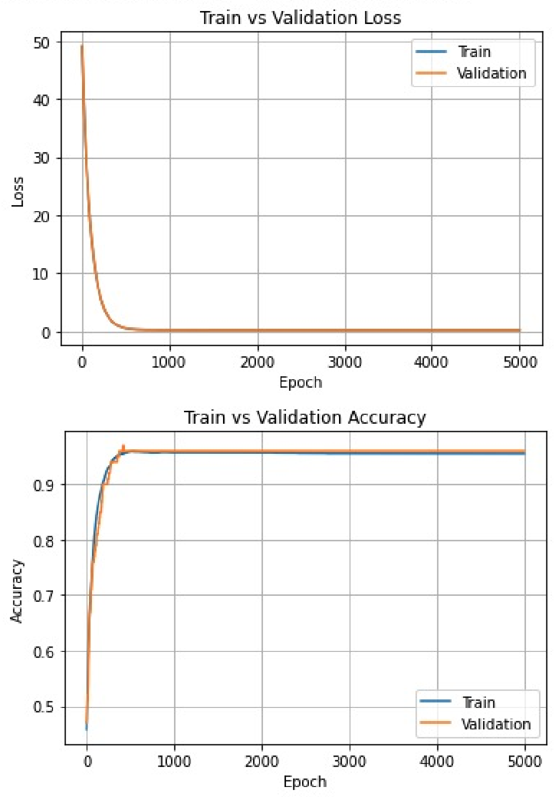
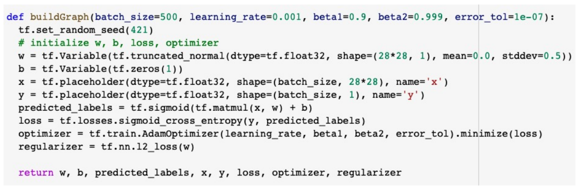


Figure 8: Training and Validation Loss and Accuracy (b=0, epochs=5000, alpha lr=0.0001, reg=0.5)

Explain: From the above 3 screenshots, we can tell that the regularization parameter of 0.1 and 0.5 results better in both accuracy and loss. However, when regularization parameter equals to 0.5, convergence is faster and the model is less over-fitting. Overall, 0.5 is the best regularization parameter among these three, resulting in training accuracy of 0.955 and validation accuracy of 0.96.

2 Logistic Regression in TensorFlow

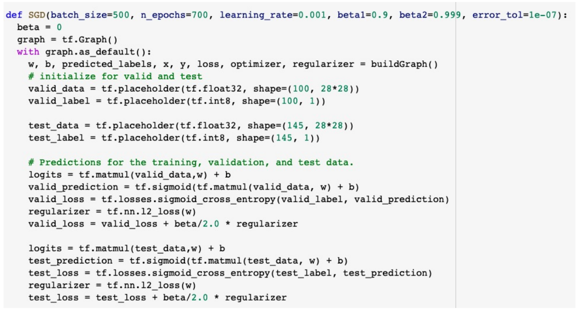
2.1 Building the Computational Graph



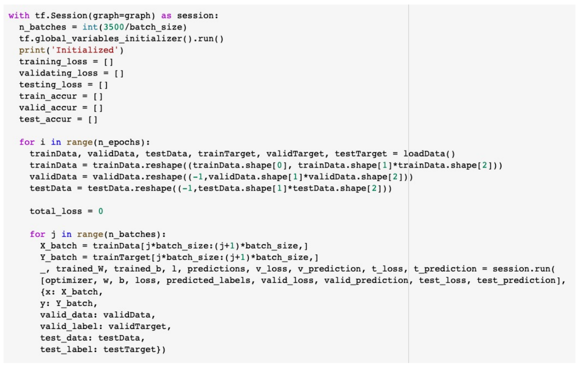
```python
def buildGraph(batch_size=500, learning_rate=0.001, beta1=0.9, beta2=0.999, error_tol=1e-07):
  tf.set_random_seed(421)
  # initialize w, b, loss, optimizer
  w = tf.Variable(tf.truncated_normal(dtype=tf.float32, shape=(28*28, 1), mean=0.0, stddev=0.5))
  b = tf.Variable(tf.zeros(1))
  x = tf.placeholder(dtype=tf.float32, shape=(batch_size, 28*28), name='x')
  y = tf.placeholder(dtype=tf.float32, shape=(batch_size, 1), name='y')
  predicted_labels = tf.sigmoid(tf.matmul(x, w) + b)
  loss = tf.losses.sigmoid_cross_entropy(y, predicted_labels)
  optimizer = tf.train.AdamOptimizer(learning_rate, beta1, beta2, error_tol).minimize(loss)
  regularizer = tf.nn.l2_loss(w)

  return w, b, predicted_labels, x, y, loss, optimizer, regularizer
```

Figure 9: Implementation of buildGraph function

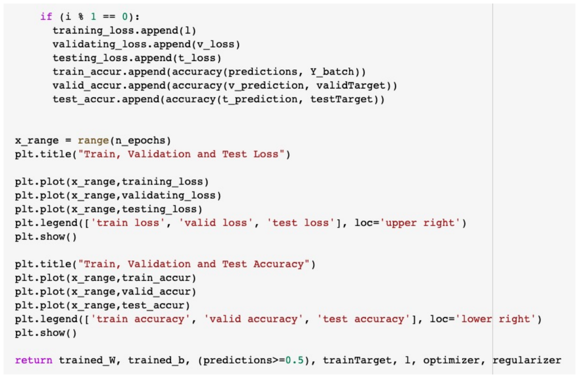
2.2 Implementing Stochastic Gradient Descent



```python
def SGD(batch_size=500, n_epochs=700, learning_rate=0.001, beta1=0.9, beta2=0.999, error_tol=1e-07):
  beta = 0
  graph = tf.Graph()
  with graph.as_default():
    w, b, predicted_labels, x, y, loss, optimizer, regularizer = buildGraph()
    # initialize for valid and test
    valid_data = tf.placeholder(tf.float32, shape=(100, 28*28))
    valid_label = tf.placeholder(tf.int8, shape=(100, 1))

    test_data = tf.placeholder(tf.float32, shape=(145, 28*28))
    test_label = tf.placeholder(tf.int8, shape=(145, 1))

    # Predictions for the training, validation, and test data.
    logits = tf.matmul(valid_data,w) + b
    valid_prediction = tf.sigmoid(tf.matmul(valid_data, w) + b)
    valid_loss = tf.losses.sigmoid_cross_entropy(valid_label, valid_prediction)
    regularizer = tf.nn.l2_loss(w)
    valid_loss = valid_loss + beta/2.0 * regularizer

    logits = tf.matmul(test_data,w) + b
    test_prediction = tf.sigmoid(tf.matmul(test_data, w) + b)
    test_loss = tf.losses.sigmoid_cross_entropy(test_label, test_prediction)
    regularizer = tf.nn.l2_loss(w)
    test_loss = test_loss + beta/2.0 * regularizer
```



```python
with tf.Session(graph=graph) as session:
  n_batches = int(3500/batch_size)
  tf.global_variables_initializer().run()
  print('Initialized')
  training_loss = []
  validating_loss = []
  testing_loss = []
  train_accur = []
  valid_accur = []
  test_accur = []

  for i in range(n_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]))

    total_loss = 0

    for j in range(n_batches):
      X_batch = trainData[j*batch_size:(j+1)*batch_size,]
      Y_batch = trainTarget[j*batch_size:(j+1)*batch_size,]
      _, trained_W, trained_b, l, predictions, v_loss, v_prediction, t_loss, t_prediction = session.run(
      [optimizer, w, b, loss, predicted_labels, valid_loss, valid_prediction, test_loss, test_prediction],
      {x: X_batch,
      y: Y_batch,
      valid_data: validData,
      valid_label: validTarget,
      test_data: testData,
      test_label: testTarget})
```



```
    if (i % 1 == 0):
      training_loss.append(l)
      validating_loss.append(v_loss)
      testing_loss.append(t_loss)
      train_accur.append(accuracy(predictions, Y_batch))
      valid_accur.append(accuracy(v_prediction, validTarget))
      test_accur.append(accuracy(t_prediction, testTarget))


x_range = range(n_epochs)
plt.title("Train, Validation and Test Loss")

plt.plot(x_range,training_loss)
plt.plot(x_range,validating_loss)
plt.plot(x_range,testing_loss)
plt.legend(['train loss', 'valid loss', 'test loss'], loc='upper right')
plt.show()

plt.title("Train, Validation and Test Accuracy")
plt.plot(x_range,train_accur)
plt.plot(x_range,valid_accur)
plt.plot(x_range,test_accur)
plt.legend(['train accuracy', 'valid accuracy', 'test accuracy'], loc='lower right')
plt.show()

return trained_W, trained_b, (predictions>=0.5), trainTarget, l, optimizer, regularizer
```

Figure 10: Implementation of SGD function

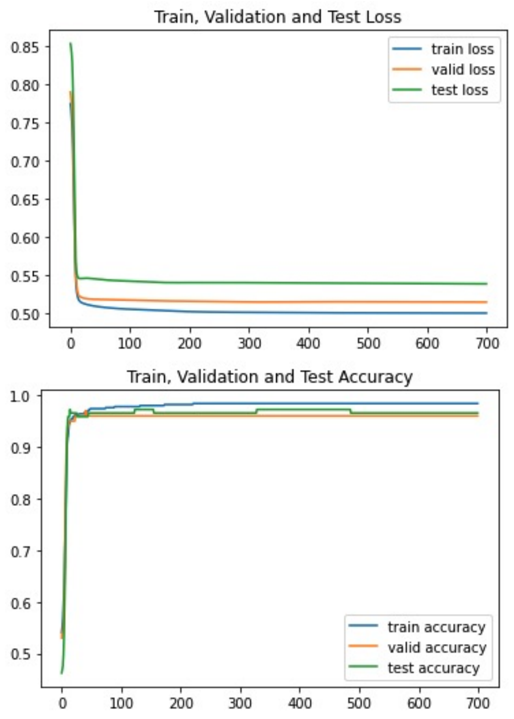


Figure 11: SGD with minibatch size of 500 and 700 epochs, alpha = 0.001, lamba = 0

2.3 Batch Size Investigation

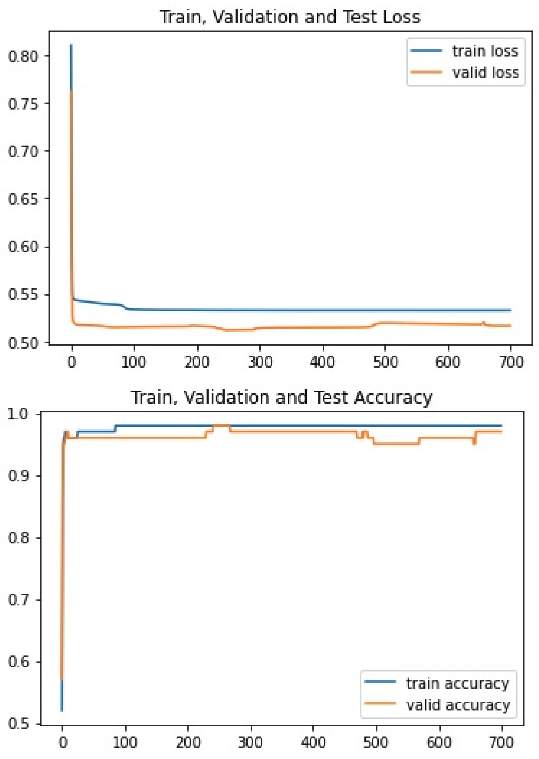


Figure 12: SGD with minibatch size of 100 and 700 epochs, alpha = 0.001, lamba = 0

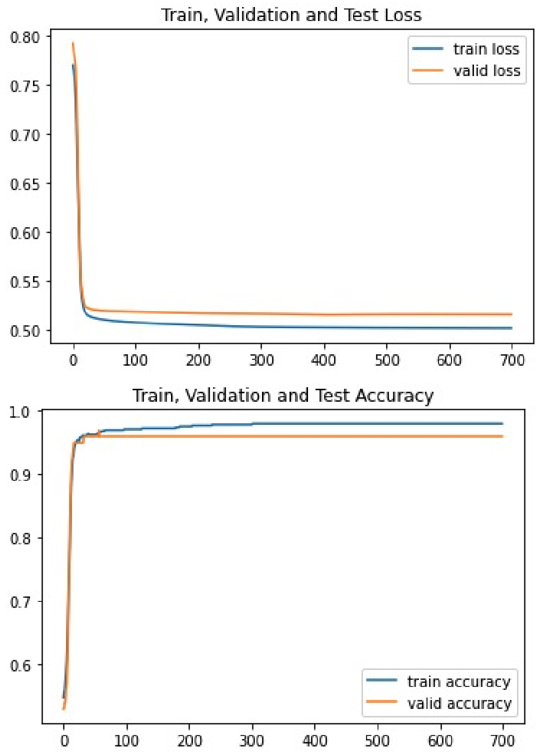


Figure 13: SGD with minibatch size of 700 and 700 epochs, alpha = 0.001, lamba = 0

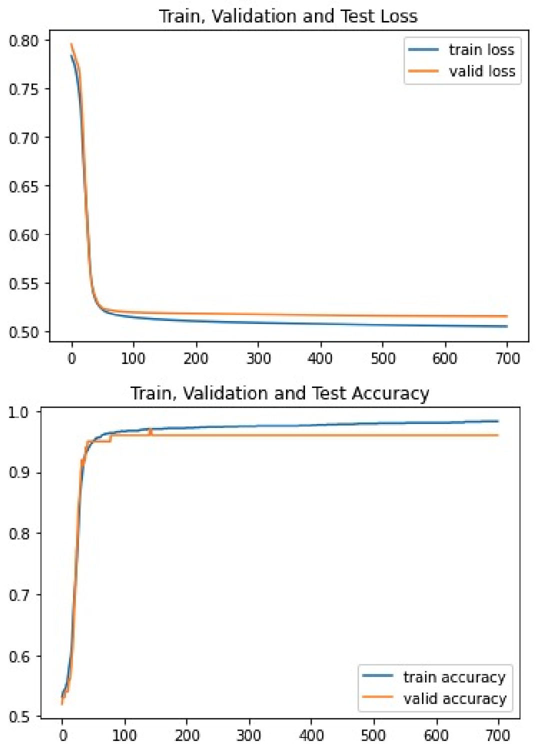
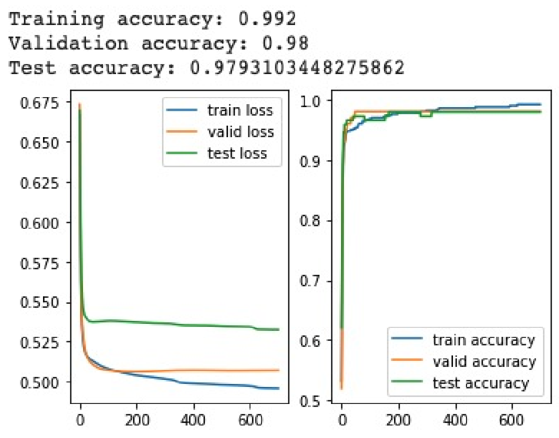


Figure 14: SGD with minibatch size of 1750 and 700 epochs, alpha = 0.001, lamba = 0

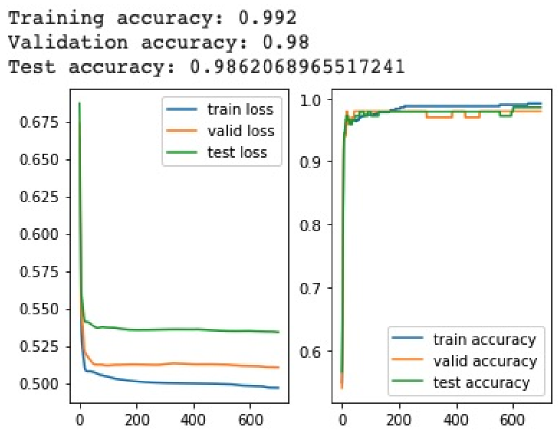
Discussion: from the 3 groups of figures above, we can tell that batch size of 100 performs the best, and batch size of 1750 performs the worst. This is because when batch size is small, more iterations can be performed in 1 epoch, resulting in higher accuracy and lower loss. However, when batch size is too small, there may be problem of overfitting.

2.4 Hyperparameter Investigation



```
Training accuracy: 0.992
Validation accuracy: 0.98
Test accuracy: 0.9793103448275862
```

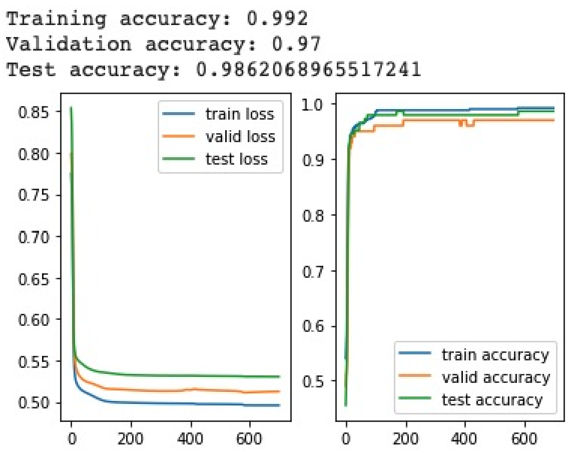
Figure 15: Adam hyperparameters: beta1=0.95



```
Training accuracy: 0.992
Validation accuracy: 0.98
Test accuracy: 0.9862068965517241
```

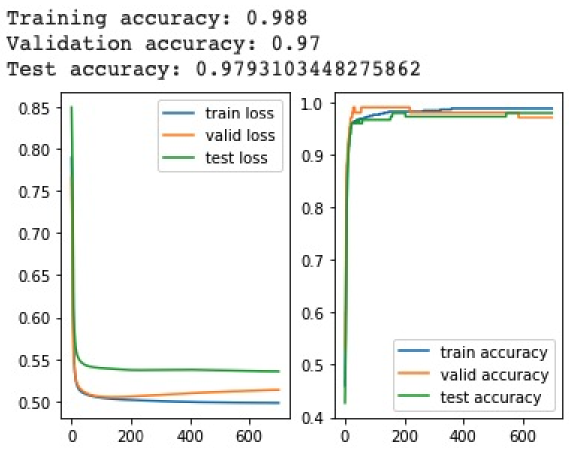
Figure 16: Adam hyperparameters: beta1=0.99

Discussion: Comparing beta1 value of 0.95 and 0.99, we noticed that lower beta1 value results in better accuracy. Beta1 is the exponential decay rate for the first momentum estimate. This is because when beta1 value is high, the model rely more on historical values, causing more fluctuation and less accuracy.



```
Training accuracy: 0.992
Validation accuracy: 0.97
Test accuracy: 0.9862068965517241
```

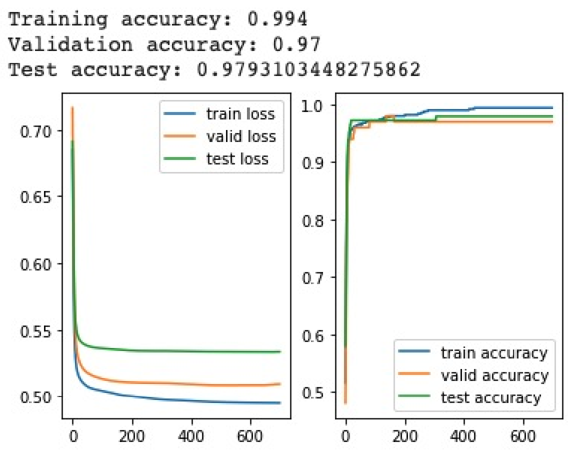
Figure 17: Adam hyperparameters: beta2=0.99



```
Training accuracy: 0.988
Validation accuracy: 0.97
Test accuracy: 0.9793103448275862
```

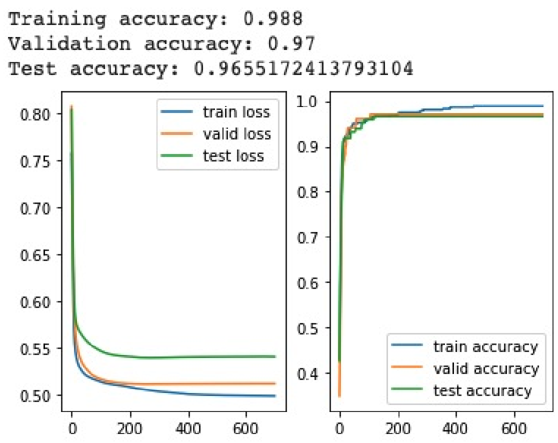
Figure 18: Adam hyperparameters: beta2=0.999

Discussion: Comparing beta2 value of 0.99 and 0.999, we noticed that lower beta2 value results better. This is similar to beta1. Beta2 is the exponential decay rate for the second momentum estimate. Lower value of beta2 will improve inconsistency in the training curve.



```
Training accuracy: 0.994
Validation accuracy: 0.97
Test accuracy: 0.9793103448275862
```

Figure 19: Adam hyperparameters: epsilon=1e-09



```
Training accuracy: 0.988
Validation accuracy: 0.97
Test accuracy: 0.9655172413793104
```

Figure 20: Adam hyperparameters: epsilon=1e-04

Discussion: Comparing epsilon value of 1e-09 and 1e-04, we noticed that lower value of epsilon results better in final accuracy. However, the training curves are less smooth than 1e-09. Epsilon is used to prevent zero division. When epsilon is small, it has less effect on weight parameters.

	$\beta_1 = 0.95$	$\beta_1 = 0.99$	$\beta_2 = 0.99$	$\beta_2 = 0.9999$	$\epsilon = 1e-09$	$\epsilon = 1e-04$
Train Accuracy	99%	99%	99%	98%	99%	98%
Validation Accuracy	98%	98%	97%	97%	97%	97%
Test Accuracy	97%	98%	98%	97%	97%	96%

Table 1: Final Train, Validation, Test Accuracy of different Adam hyperparameters

2.5 Comparison against Batch GD

	Batch Gradient Descent	SGD with Adam
Train Accuracy	96%	99%
Validation Accuracy	97%	98%
Test Accuracy	97%	98%

Table 2: Comparing accuracy between Batch Gradient Descent and SGD

Discussion: Comparing three accuracy across these two methods, they both performs well with tuned parameters. While SGD can do slightly better than BGD in terms of final values, SGD is also nosier than BGD in both loss and accuracy plots. The reason of SGD performs slightly better may be due to 2 reasons: 1. Adam optimizer can help model overcome local minimums and 2. help to increase convergence speed. In addition, the reason why two methods all reach satisfying results may be due to the employment of CE loss function.