# ECE421 Assignment 1

## Part1 Logistic Regression with Numpy

### Q1 Loss Function and Gradient

Loss function derivation:

$$\hat{y}(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b),$$

$$\sigma(z) = 1/(1 + \exp(-z))$$

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

```
# sigma(w*x+b)
def sigma(w, b, x):
    z = np.matmul(x, w) + b
    #print("z",z)
    sig = 1/(1 + np.exp(-z))
    return sig

# L = Lce + Lw
def loss(w, b, x, y, reg):
    y_hat = sigma(w, b, x) #3500*1
    Lce = np.sum(-y*np.log(y_hat) - (1 - y)*np.log(1 - y_hat)) / len(y)
    #print("Lce",Lce)
    Lw = (np.linalg.norm(w)**2)*reg/2
    #print("Lw",Lw)
    return Lce + Lw
```

Gradient Loss function:

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

Let $z = w^T x + b$ $\qquad L\,inside = -y\,log\,\hat{y} - (1-y)log\,(1-\hat{y})$

$$\frac{d\,L\,inside}{d\,\hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \qquad \frac{d\hat{y}}{dz} = \hat{y}(1-\hat{y}) \qquad \frac{dz}{dw} = x \qquad \frac{dz}{db} = 1$$

$$\frac{d\,Lce}{d\,w} = \frac{1}{N}\sum_{n=1}^{N}\left(\frac{d\,Lce}{d\,\hat{y}} * \frac{d\hat{y}}{dz} * \frac{dz}{dw}\right) = \frac{1}{N}\left[\,x^T(\hat{y} - y)\right] \qquad \frac{d\,Lw}{d\,w} = \lambda w$$

$$\frac{d\,Lce}{d\,b} = \frac{1}{N}\sum_{n=1}^{N}\left(\frac{d\,Lce}{d\,\hat{y}} * \frac{d\hat{y}}{dz} * \frac{dz}{db}\right) = \frac{1}{N}\left[\,(\hat{y} - y)\right] \qquad \frac{d\,Lw}{d\,b} = 0$$

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

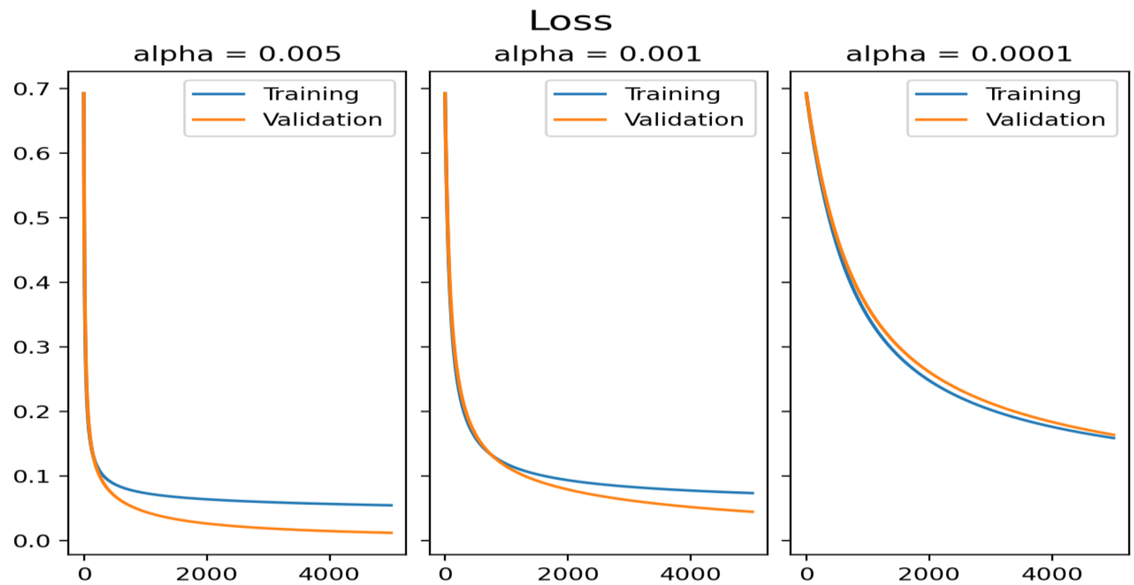
```
def grad_loss(w, b, x, y, reg):
    y_hat = sigma(w, b, x)
    diff_w = np.matmul(np.transpose(x), (y_hat - y))/(np.shape(y)[0]) + reg*w
    diff_b = np.sum(y_hat - y, keepdims=True) / (np.shape(y)[0])

    return diff_w, diff_b
```

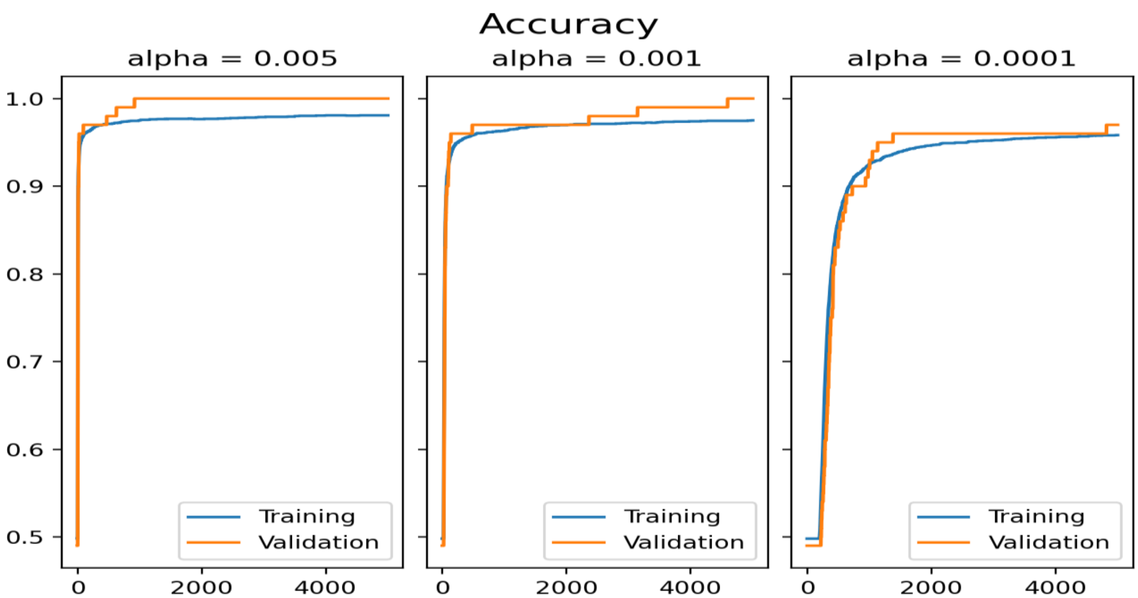
## Q2 Gradient Descent Implementation

The gradient descent function implemented will calculate the new weights and bias in every iteration of epochs. Also the loss and accuracy data in each iteration will be stored to the global variable loss_array and accu_array

```
# Global variable to store the changes of loss
loss_array = []
accu_array = []
def grad_descent(w, b, x, y, alpha, epochs, reg, error_tol = 1e-7):
    global loss_array
    global accu_array
    loss_array = []
    loss_array.append(loss(w, b, x, y, reg))
    accu_array = []
    out_train = np.matmul(x,w)+b
    accur = [np.sum((out_train>=0.5)==y)/(x.shape[0])]
    accu_array.append(accur)

    #Stop when total number of epochs reached
    for i in range(epochs):
        grad_w, grad_b = grad_loss(w, b, x, y, reg)
        new_w = w - alpha * grad_w
        new_b = b - alpha * grad_b
        # calculate the new loss value
        loss_array.append(loss(new_w, new_b, x, y, reg))
        # calculate the new accuracy value
        out_train = np.matmul(x,new_w)+new_b
        accur = [np.sum((out_train>=0.5)==y)/(x.shape[0])]
        accu_array.append(accur)
        # check norm(new_w - w)< error_tol
        norm_err = np.linalg.norm(new_w - w)
        if norm_err < error_tol:
            return new_w,new_b
        else:
            w = new_w
            b = new_b
    return w,b
```

# Q3 Tuning the Learning Rate



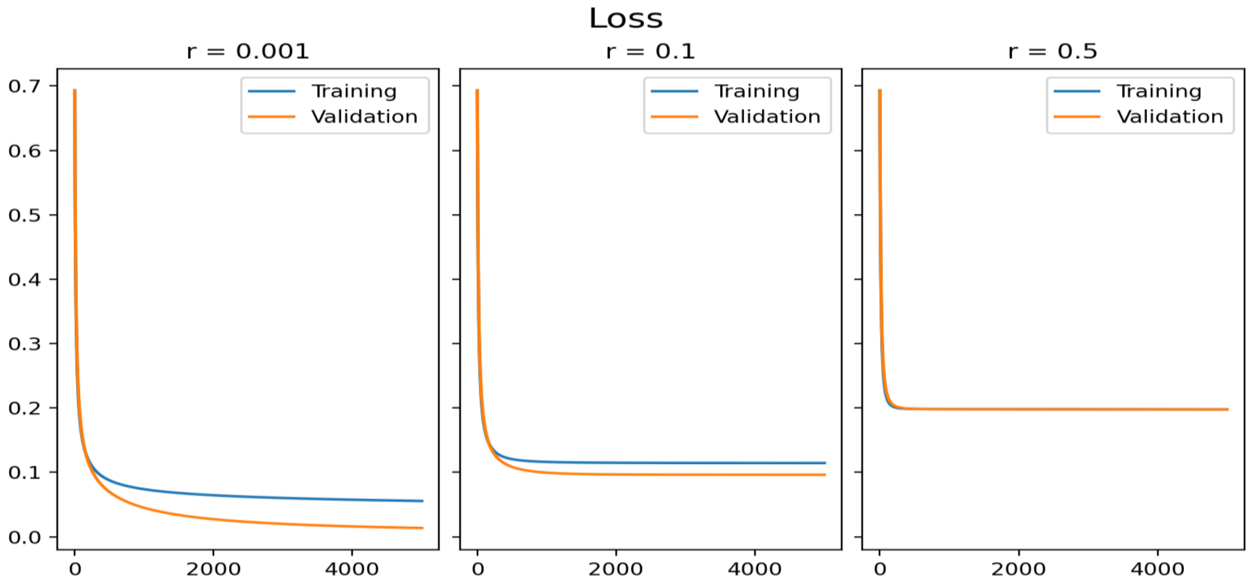

| | alpha = 0.005 | Alpha = 0.001 | Alpha = 0.0001 |
|---|---|---|---|
| Training Accuracy | 0.9814 | 0.9758 | 0.9595 |
| Validation Accuracy | 0.9991 | 0.9751 | 0.9722 |

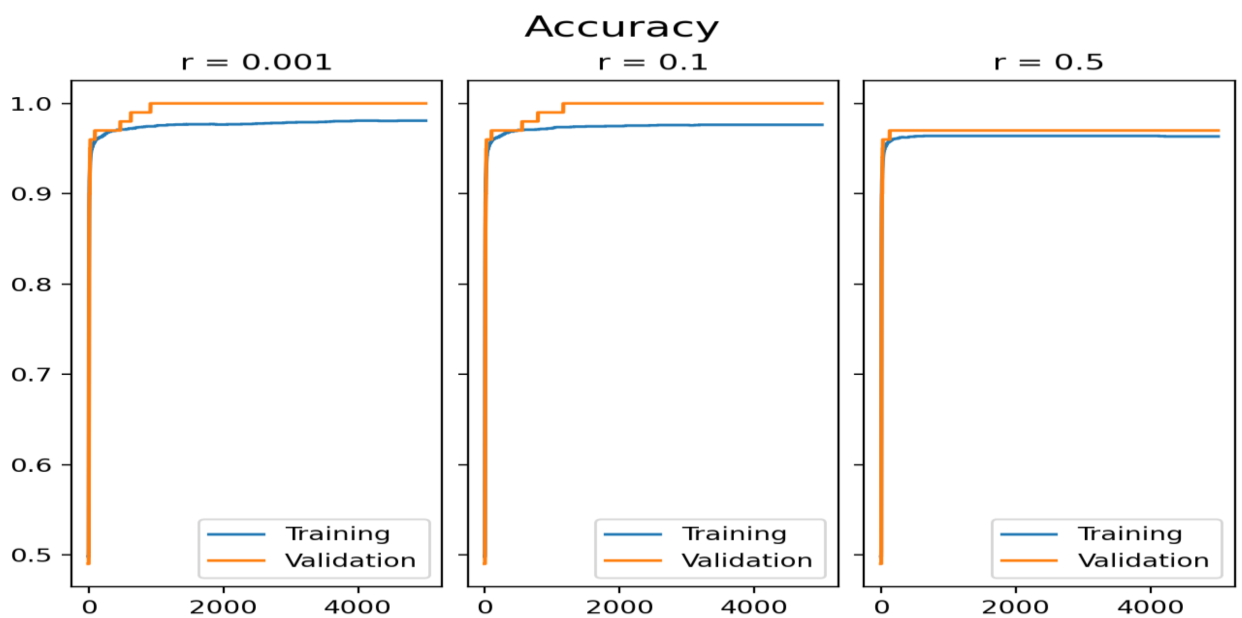
Table above shows the Training and Validation Accuracy after 5000 epochs.

From figure above, the higher the learning rate, the faster the training and validation loss will decrease.

Thus, given limited epochs the larger learning rate will result in higher accuracy. Hereby alpha = 0.005 provides the highest accuracy.

## Q3 Generalization




| | r = 0.001 | r = 0.1 | r = 0.5 |
|---|---|---|---|
| Training Accuracy | 1.0019 | 0.9786 | 0.9616 |
| Validation Accuracy | 0.9836 | 0.9772 | 0.9680 |

Table above shows the Training and Validation Accuracy after 5000 epochs.

As we can see with increasing regularization parameter the accuracy is decreasing, but the difference between the training accuracy and validation accuracy is also decreasing.Thus regularization is to prevent the model from over-fitting on the actual data.

# Part2  Logistic Regression in TensorFlow

## Q1 Building the Computational Graph

```
def buildGraph(learning_rate, b1=None, b2=None, eps=None):

  #Initialize weight and bias tensor
  W = tf.Variable(tf.truncated_normal(shape=(784, 1), mean=0.0, stddev=0.5, dtype=tf.float32, seed =None, name=None))
  b = tf.Variable(tf.zeros(1))

  #Place holder for Data, Label, regularization parameter
  x = tf.placeholder(tf.float32, shape=(None, 784))
  y = tf.placeholder(tf.float32, shape=(None, 1))
  reg = tf.placeholder(tf.float32, shape=None)

  #The loss tensor
  z = tf.matmul(x,W)+b
  y_hat = tf.nn.sigmoid(z)
  lce = tf.losses.sigmoid_cross_entropy(y, y_hat)
  lw = reg/2.0 * tf.nn.l2_loss(W)
  loss = lce + lw
```

```
#Adam optimizer
'''
tf.compat.v1.train.AdamOptimizer(
  learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08, use_locking=False,
  name='Adam')
'''
if(b1):
  optimizer = tf.train.AdamOptimizer(learning_rate,beta1=b1).minimize(loss)
elif(b2):
  optimizer = tf.train.AdamOptimizer(learning_rate,beta2=b2).minimize(loss)
elif(eps):
  optimizer = tf.train.AdamOptimizer(learning_rate,epsilon=eps).minimize(loss)
else:
  optimizer = tf.train.AdamOptimizer(learning_rate).minimize(loss)

#The function should return the TensorFlow objects for
#weight, bias, predicted labels, real labels, the loss, and the optimizer.
return W, b, x, y, y_hat, loss, optimizer, reg
```

## Q2 Implementing Stochastic Gradient Descent

Helper function to randomly shuffle the Traindata and Traintarget

```
def random_shuffle(trainData,trainTarget):
  # randomly shuffle traindata, trainTarget
  order = np.random.permutation(len(trainTarget))
  #print("trainData shape",trainData.shape)
  return trainData[order], trainTarget[order]
```

Helper function for loading and processing data

```
def Process_Data():
    # 3500*28*28 100*28*28  145*28*28
    #[trainData, validData, testData, trainTarget, validTarget, testTarget]
    dataList = loadData()
    dataList = list(dataList)

    # 3500*784    100*784    145*784
    # [trainData, validData, testData]
    for i, data in enumerate(dataList[:3]):
        dataList[i] = data.reshape(len(data), -1)

    trainData  = dataList[0] #3500*784
    validData  = dataList[1] #100*784
    testData   = dataList[2] #145*784

    trainTarget= dataList[3].astype(int) #3500*1
    validTarget= dataList[4].astype(int) #100*1
    testTarget = dataList[5].astype(int) #145*1

    return trainData,validData,testData,trainTarget,validTarget,testTarget
```

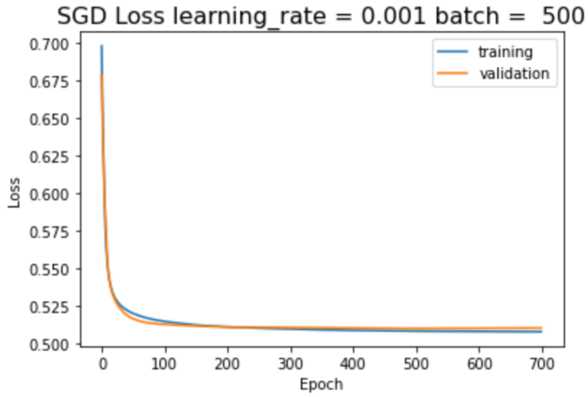
SGD function

```
def SGD(epochs, batch_size, learning_rate):

  #========================== Load and Proccess Data ================================#
  # trainData, trainTarget, validData, validTarget, testData, testTarget = loader()
  # 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]))

  #3500*784 100*784 145*784  3500*1  100*1    145*1
  trainData,validData,testData,trainTarget,validTarget,testTarget = Process_Data()

  # build graph
  W, b, x, y, y_hat, loss, optimizer, reg = buildGraph(learning_rate)

  #Variables to be calculated
  regularization_parameter = 0
  train_loss, valid_loss = [],[]
  train_accur, valid_accur = [],[]
  test_loss, test_accur = [],[]
  n_batch = int(len(trainTarget)/batch_size)
```

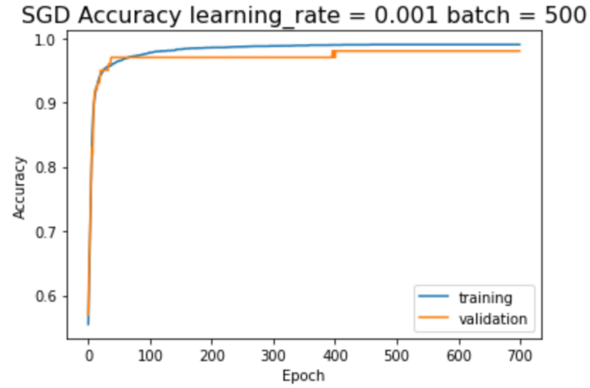
```python
# start training
with tf.Session() as ssess:
    ssess.run(tf.global_variables_initializer())
    for i in range(epochs):
        trainData,trainTarget = random_shuffle(trainData,trainTarget)
        loss_train = 0
        accur_train = 0
        # train modle from batches
        for n in range(n_batch):
            # Training Data Divide into batch
            batch_X = trainData[n*batch_size: (n+1)*batch_size]
            batch_Y =  trainTarget[n*batch_size: (n+1)*batch_size]
            new_y_hat, new_loss, new_y, op = ssess.run([y_hat, loss, y, optimizer],
                                                        feed_dict={ x: batch_X,
                                                                   y: batch_Y,
                                                                   reg: regularization_parameter})
            #calculate training acuuracy
            train_a = np.sum((new_y_hat>0.5)==new_y)/batch_size#accuracy
            loss_train+=new_loss
            accur_train+=train_a

        train_loss.append(loss_train/n_batch)
        train_accur.append(accur_train/n_batch)
```

```python
        #==================================== Validation Data ======================================#
        Valid_predic, Valid_loss, new_y = ssess.run([y_hat, loss, y],
                                                    feed_dict={x: validData,
                                                               y: validTarget,
                                                               reg: regularization_parameter})
        valid_loss.append(Valid_loss)
        valid_accur.append(np.sum((Valid_predic>0.5)==new_y)/new_y.shape[0])

        #==================================== Validation Data ======================================#
        Test_predic, Test_loss, new_y = ssess.run([y_hat, loss, y],
                                                  feed_dict={x: testData,
                                                             y: testTarget,
                                                             reg: regularization_parameter})
        test_loss.append(Test_loss)
        test_accur.append(np.sum((Test_predic>0.5)==new_y)/new_y.shape[0])

    plot_loss(train_loss, valid_loss, learning_rate, batch_size)
    plot_Accu(train_accur, valid_accur, learning_rate, batch_size)
return train_loss, valid_loss, train_accur, valid_accur, test_accur
```

```
train_loss is  0.5078967767102378
valid_loss is  0.5099594
```

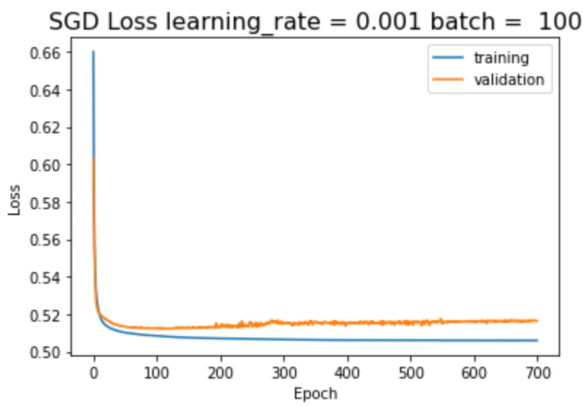



```
train_accuracy is  0.9900000000000001
valid_accuracy is  0.98
```

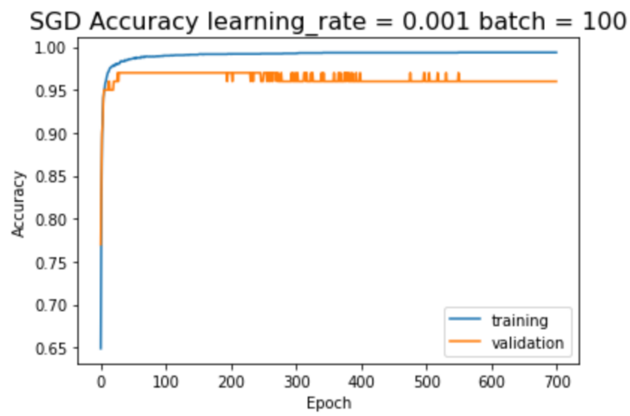



# Q3 Batch Size Investigation

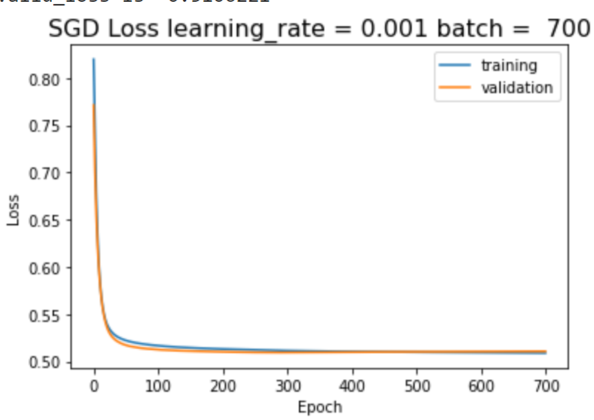
```
train_loss is  0.5059379356248038
valid_loss is  0.5162749
```



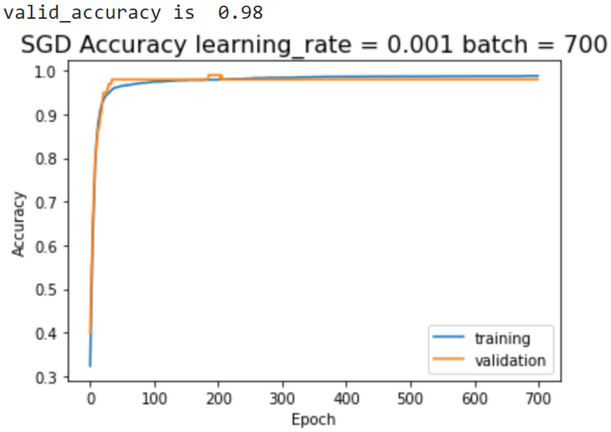

```
train_accuracy is  0.9937142857142856
valid_accuracy is  0.96
```



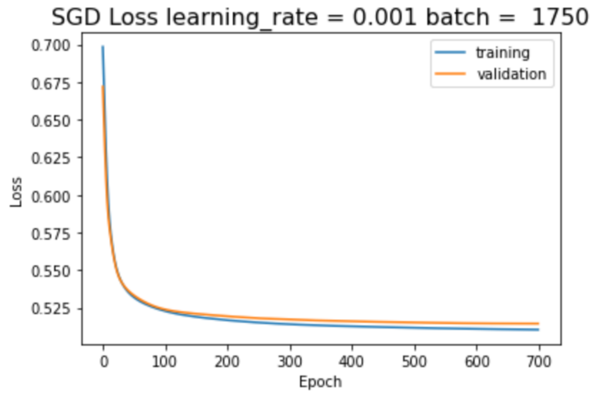

```
train_loss is  0.5088092505931854
valid_loss is  0.5106221
```



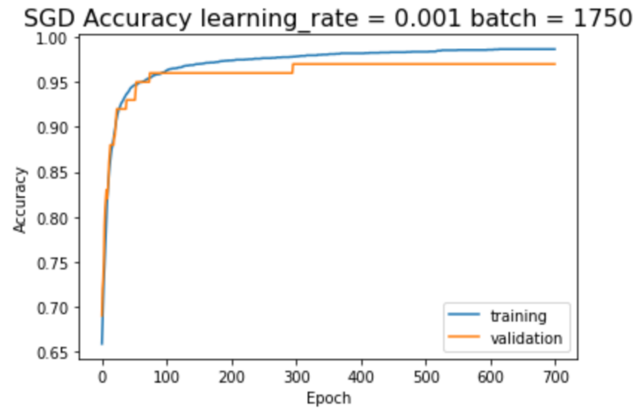

```
train_accuracy is  0.9882857142857142
valid_accuracy is  0.98
```



```
train_loss is  0.5103777050971985
valid_loss is  0.5144537
```




```
train_accuracy is  0.9865714285714287
valid_accuracy is  0.97
```




| | Batch = 100 | Batch = 700 | Batch = 1750 |
|---|---|---|---|
| Training Accuracy | 0.994 | 0.988 | 0.9865 |
| Validation Accuracy | 0.96 | 0.98 | 0.97 |

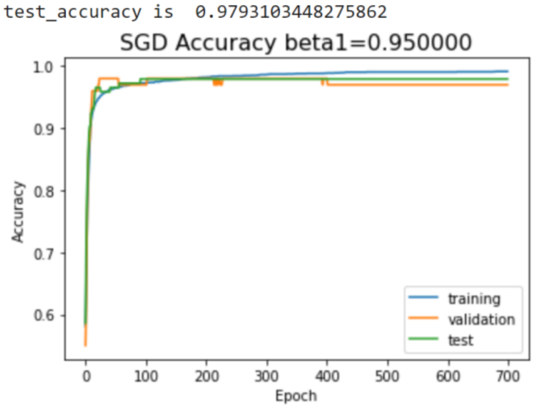
Both the Training and Validation accuracy is high with 3 different batch size.

It seems that batch size will post no significant impact to the ultimate accuracy.

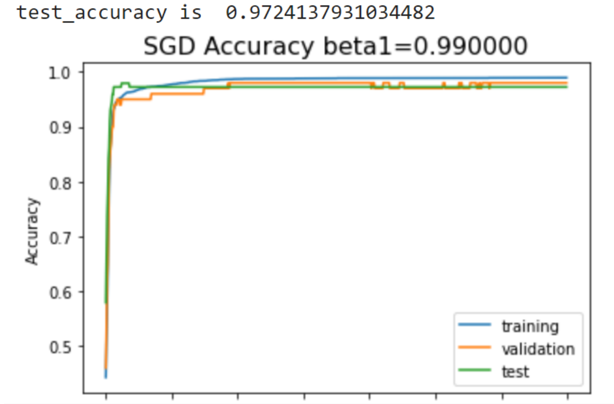
But from the graph, smaller batch size will cause the accuracy curve to vibrate , while with large batch size the accuracy increase steadily.

## Q4 Hyperparameter Investigation

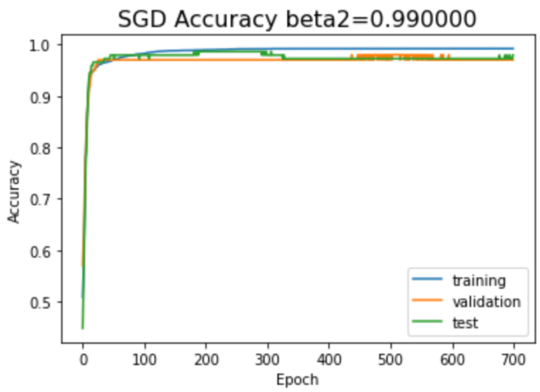
```
train_accuracy is  0.9914285714285713
valid_accuracy is  0.97
test_accuracy is  0.9793103448275862
```



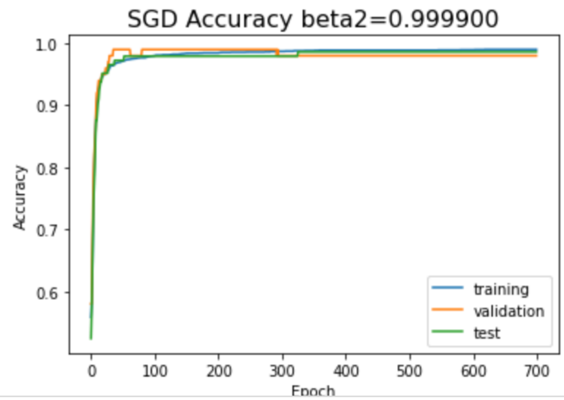

```
train_accuracy is  0.9894285714285714
valid_accuracy is  0.98
test_accuracy is  0.9724137931034482
```



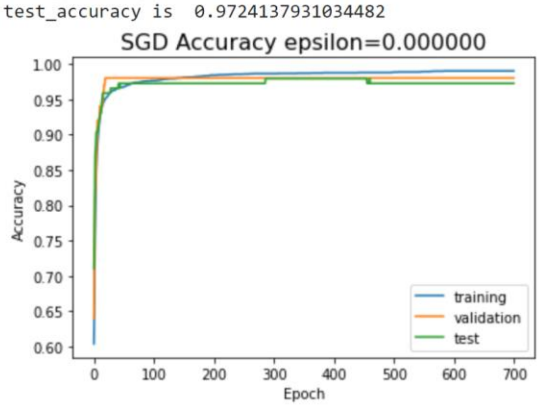
```
train_accuracy is  0.9919999999999999
valid_accuracy is  0.97
test_accuracy is  0.9793103448275862
```



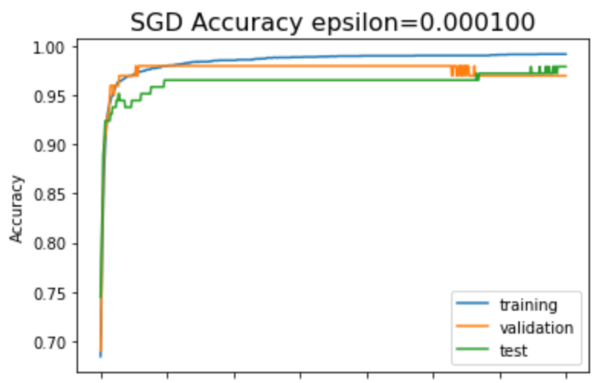

```
train_accuracy is  0.9899999999999999
valid_accuracy is  0.98
test_accuracy is  0.9862068965517241
```




```
train_accuracy is  0.9900000000000001
valid_accuracy is  0.98
test_accuracy is  0.9724137931034482
```

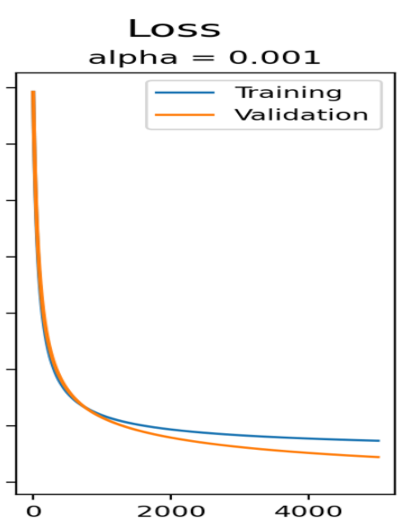


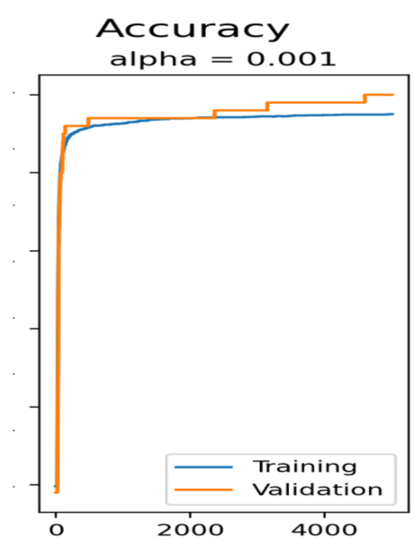

```
train_accuracy is  0.9920000000000001
valid_accuracy is  0.97
test_accuracy is  0.9793103448275862
```




# Q5 Comparison against Batch GD

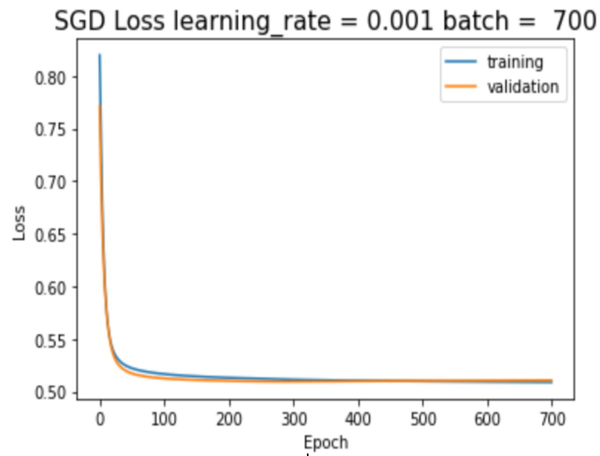
# Batch GD with alpha = 0.001



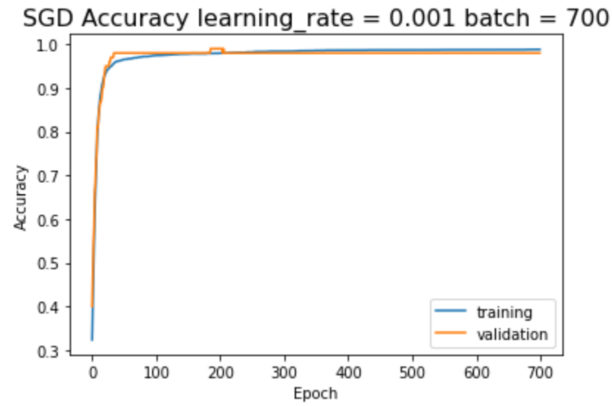




| Accuracy | Alpha = 0.001 |
|---|---|
| Training | 0.9758 |
| Validation | 0.9751 |

```
train_loss is  0.5088092505931854
valid_loss is  0.5106221
```




```
train_accuracy is  0.9882857142857142
valid_accuracy is  0.98
```




| Accuracy | Batch = 700 |
|---|---|
| Training | 0.988 |
| Validation | 0.98 |

From graph we can see that SGD use 700 epochs to achieve accuracy of 0.98, while Batch GD use 5000 epoches to achieve the accuracy of 0.97. Thus the SGD is much faster and have better performance than the Batch GD.