ECE421 Introduction to Machine Learning
Assignment #1
Linear and Logistic Regression

Yan, Xuanming
[redacted]
Contribution: 50%

Wang, Chu Qing
[redacted]
Contribution: 50%

Jan 2019

1 Linear Regression

Mean Squared Error (MSE) loss function $\mathcal{L}_\mathcal{D}$ and weight decay loss $\mathcal{L}_W$ for training a linear regression model,

$$\begin{aligned}\mathcal{L} &= \mathcal{L}_\mathcal{D} + \mathcal{L}_\mathbf{W} \\ &= \sum_{n=1}^{N} \frac{1}{2N} \left\| \mathbf{W}^T x^{(n)} + b - y^{(n)} \right\|_2^2 + \frac{\lambda}{2} \|\mathbf{W}\|_2^2\end{aligned}$$

1.1 Loss Function and Gradient

MSE written in Python is trivial,

```
def MSE(W, b, x, y, reg):
    error = np.matmul(x,W) + b - y
    mse = (np.sum(error*error))/((2*np.shape(y)[0])) + reg/2*np.sum(W*W)
    return mse
```

Listing 1: MSE

From above MSE loss function $\mathcal{L}_\mathcal{D}$ and weight decay loss $\mathcal{L}_W$, we can express as:

$$\mathcal{L}(\mathbf{W}) = \frac{1}{2N} \sum_{n=1}^{N} E_n(\mathbf{W})^2 + \lambda \sum_{i=1}^{N} w_i^2$$

Where $E_n(\mathbf{W}) = (\sum_{i=1}^{N} w_i x_i^{(n)}) + b - y^{(n)}$. The goal is to choose $w_1, w_2, \ldots, w_D$ and b to minimize $\mathcal{L}$. In order to solve the optimization problem, we'll need to use partial derivatives. Take derivatives of $\mathcal{L}$ with respect to each element's weight, we have:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_i} &= \frac{1}{N}(\sum_{n=1}^{N} E_n(a) x_i^{(n)}) + 2\lambda w_i \\ &= \frac{1}{N} \sum_{n=1}^{N} x_i^{(n)} (\sum_{i'} w_{i'} x_{i'}^{(n)} + b - y^{(n)}) + 2\lambda w_i\end{aligned}$$

Take gradient of $\mathcal{L}$ with respect to bias, we have:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= \frac{1}{N}(\sum_{n=1}^{N} E_n(a) x_i^{(n)}) \\ &= \frac{1}{N} \sum_{n=1}^{N} (\sum_{i'} w_{i'} x_{i'}^{(n)} + b - y^{(n)})\end{aligned}$$

Implement above equation in matrix from:

```python
def gradMSE(W, b, x, y, reg):
    error = np.matmul(x,W) + b - y
    grad_mse_W = np.matmul(np.transpose(x),error)/(np.shape(y)[0]) + 2*reg*W
    grad_mse_b = (np.sum(error))/(np.shape(y)[0])
    return grad_mse_W, grad_mse_b
```

Listing 2: gradMSE

1.2 Gradient Descent Implementation

Gradient Descent is one of the most trivial way for optimization. It minimizes cost function in an iterative approach. To do gradient descent, we initialize the weight and bias using Gaussian distribution with mean 0 and variance 1, and repeatedly adjust them in the direction of most decreases the cost function. We repeat this procedure until the iterates converge, or stop changing much.

Gradient is just the direction of steepest ascent of a function. The entries of the gradient vector are simply the partial derivatives with respect to each of the variables:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_i} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial w_D} \end{bmatrix}$$

Then we can formalize this using the update rule shown below, which is Gradient Descent:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{W}}$$

The constant α is known as a learning rate. The larger it is, the larger a step we take.

```python
def grad_descent(W, b, trainingData, trainingLabels, alpha, iterations, reg, EPS):
    for i in range(iterations):
        grad_mse_W, grad_mse_b = gradMSE(W, b, trainingData, trainingLabels, reg)
        new_W = W - alpha * grad_mse_W
        new_b = b - alpha * grad_mse_b
        mag = np.linalg.norm(new_W-W)
        if mag<EPS:
            return new_W,new_b
        else:
            W = new_W
            b = new_b
    return W,b
```

Listing 3: grad descent

1.3 Tuning the Learning Rate

This table below shows final accuracy after 5000 epochs,

$\lambda = 0$	$\alpha = 0.005$	$\alpha = 0.001$	$\alpha = 0.0001$
Training data accuracy	0.644	0.5771428571428572	0.5225714285714286
Valid data accuracy	0.62	0.58	0.54
Test data accuracy	0.6137931034482759	0.47586206896551725	0.5103448275862069

Table 1: Training, validation and test losses with respect to different learning rate.

We initialize the weight and bias using Gaussian distribution with mean 0 and and standard deviation 1, and $\lambda = 0$. Fig 1 shows the learning curves and accuracy using different learning rates among training, validation and test sets. Table 1 shows the final accuracy. According to these data, we could easily conclude that the model performs the best when $\alpha = 0.005$. It provides the highest accuracy and lowest loss on both training and validation sets and also converges faster than the other two. We also observe that $\alpha = 0.0001$ produces a low quality result comparing to the other two because it makes very slow as approaching the global minimum.

Thus, since all three models have looked through the same amount of training data (5000 epochs), a relatively large training rate is a good choice in order to achieve convergence faster. One thing we noticed from this model is that we cannot reach very high accuracy if we train this model only 5000 epochs. Therefore, We have also performed experiment

with very large training epoch (100000 epochs) to debug the model, and found that there was no significant sign of over-fitting(but we can still notice that validation and test sets accuracy are lower to training set). This is most likely due to the fact that the model is not complex enough to fully over-fit on the full training data-set.

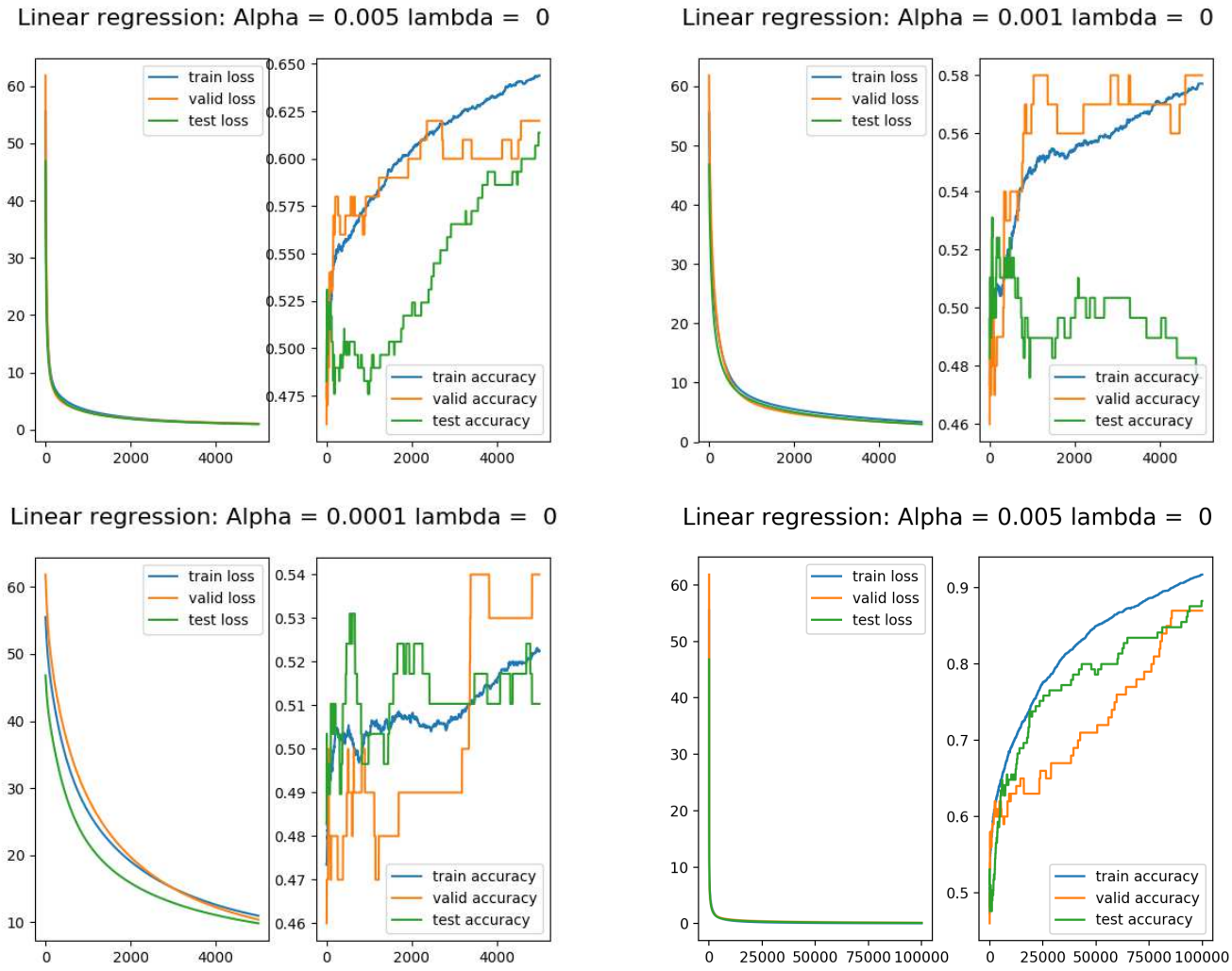

Figure 1: Training, validation and test losses with respect to different learning rate.

To summarize, with smaller learning rate, classifier will take more epochs to converge. Therefore, smaller learning rate will take more time to train. However, larger learning rate can over shoot.

1.4 Generalization

Fig 2 shows the accuracy using different weight decays among training, validation and test sets. Table 2 shows the final accuracy on training, validation and test sets. The purpose of introducing weight decay in the model is to prevent the weight from growing too large over-fitting on the training data.

From previous task, we noticed that with $\alpha = 0.005$, training accuracy is higher than validating and testing accuracy. Which shows the sign of over-fitting. In order to address over-fitting, we can manually tune the weight decay value, which is λ in this problem. We could see as λ increases, the training loss increases, which is expected because to prevent over-fitting means to prevent extremely good fitting on training set. However, small λ value in this model contributes little for preventing over-fitting and improve convergence speed. When $\lambda = 0.1$, we have lowest valid/test data loss and highest valid/test data accuracy, but model took about 2000 epochs to converge. Compare to $\lambda = 0.5$, which gives relatively same performance, it only took less than 500 epochs to converge. Therefore, we conclude that, $\lambda = 0.5$ provides a faster convergence and lower validation loss; therefore, we would choose $\lambda = 0.5$ as our best weight decay parameter.

This table below shows final accuracy after 5000 epochs,

$\alpha = 0.005$	$\lambda = 0.001$	$\lambda = 0.1$	$\lambda = 0.5$
Training data accuracy	0.6508571428571429	0.9817142857142858	0.9771428571428571
Valid data accuracy	0.62	0.99	0.98
Test data accuracy	0.6275862068965518	0.9862068965517241	0.9793103448275862

Table 2: Training, validation and test losses with respect to different weight decays.

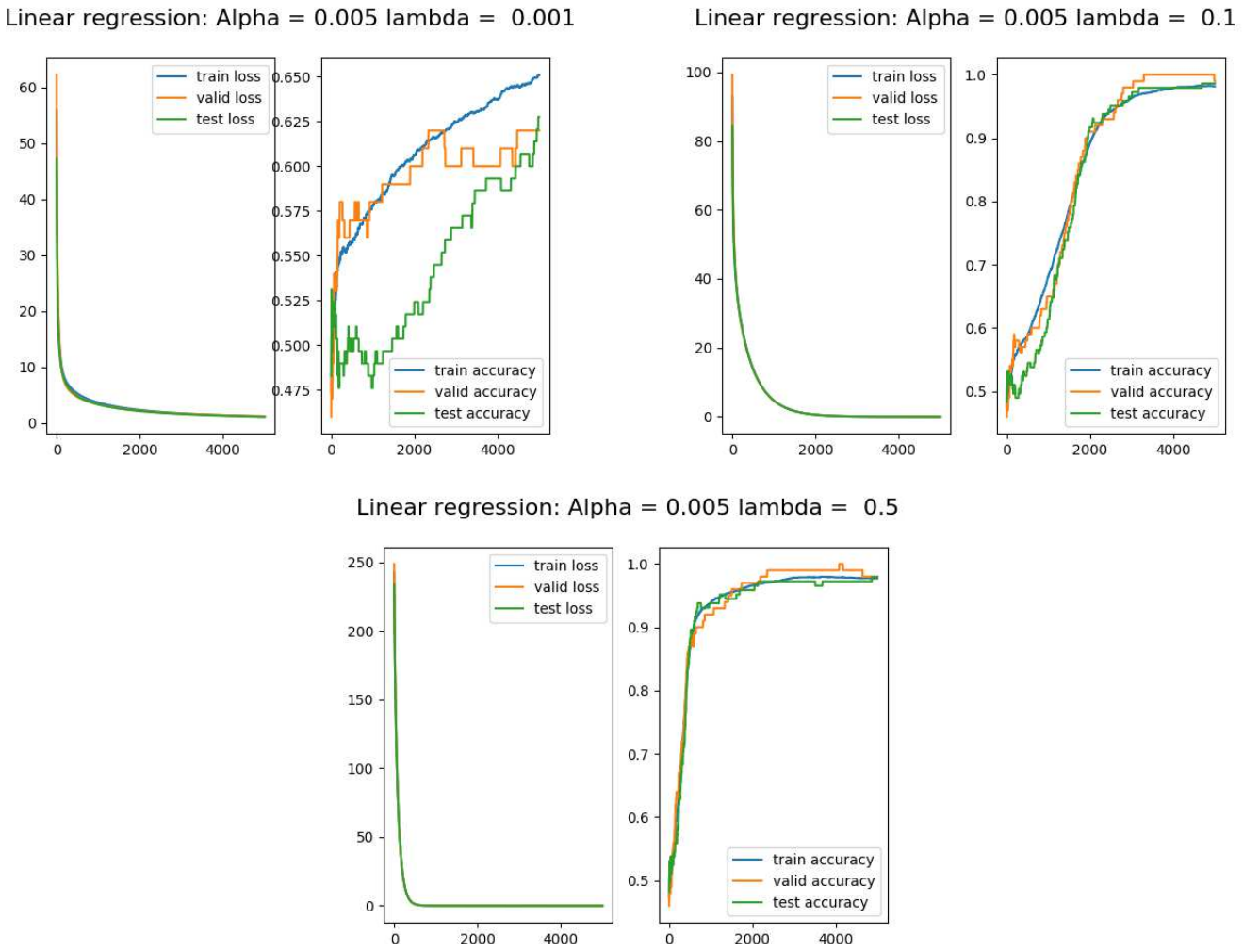

Figure 2: Training, validation and test losses with respect to different regularization.

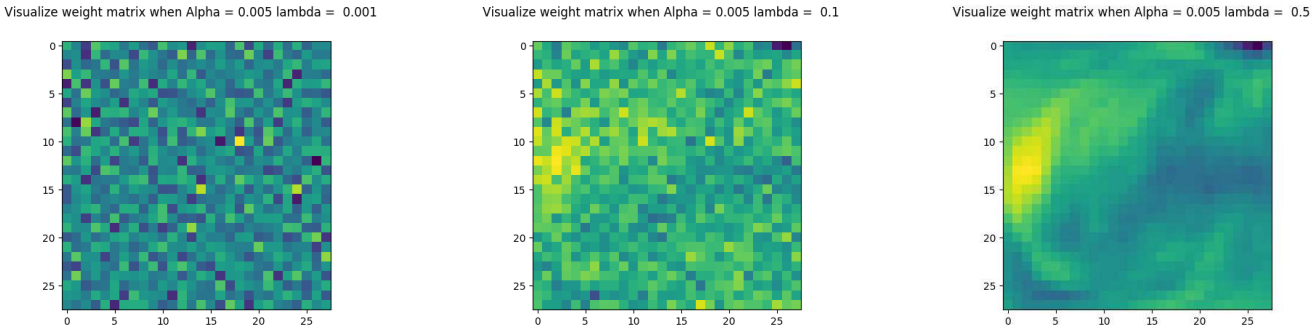

Figure 3: Visualization of Weight matrix with different regularization.

We can also visualize weight matrix to compare the performance of each model. Fig 3 shows visualized weight matrix under different weight decay values. when $\lambda = 0.5$, we can easily recognize the letter "C", this shows that after model is trained, our classifier captured the most important features for this data-set. Similar to "eigenfaces" in computer vision problem of human face recognition.

To summaries, the meaning of regularization is to prevent the model from over-fitting on the actual data; therefore, it should sacrifice some training accuracy and improve the validation/test accuracy. The purpose of validation set is to better tune the hyper-parameters such as learning rate, weight decay, and network architecture to find the best possible model. And test set is supposed to represent the real data, which shouldn't be visible during the phase while we train and tune the model. Otherwise, the test set becomes another validation set and introduces biases while selecting the model.

1.5 Comparing Batch GD with normal equation

Table 3 below shows final training MSE, accuracy and computation time between Batch GD and normal equation. We could easily tell Normal Equation performs much better than Batch GD with zero weight decay, which is expected because the analytic solution is the optimum. Batch GD model takes about half minute to training but the Normal Equation takes less than a second to compute, Batch GD is far behind Normal Equation in this scenario.

But is is not always the case. In order to find out under what circumstances Batch GD would perform better, we need to analyze these two approaches first. If the data set contains N samples and each sample has d features, the complexity of Normal Equation is $\mathbf{O}(Nd^2)$, where inverting a matrix is very expensive. It may have complexity of $\mathbf{O}(d^3)$(even though there are fancy modern numerical techniques for matrix inversion may have better complexity, it still above quadratic). And the complexity of Batch GD is $\mathbf{O}(Nd \cdot iteration)$. If $N \gg d$, Batch GD would be more efficient when the data set has a large amount of data samples; otherwise, Normal Equation would produce a better solution in a quicker way.

	Running Time	Train data accuracy	Valid data accuracy	Test data accuracy	MSE
Normal equation	0.36765 second	99%	96%	94%	0.0094
Batch GD	20.93 seconds	64.4%	62%	61.37%	1.013

Table 3: Comparing Batch GD with normal equation.

```
def norm_equation(x, y):
    x_b = np.ones((np.shape(x)[0],1))
    xx = np.hstack((x,x_b))
    W = np.matmul(np.matmul(np.linalg.inv(np.matmul(np.transpose(xx), xx)), np.transpose(xx)), y)
    print(W.shape)
    return W[:-1,:],W[-1][0]
```

Listing 4: Numpy script for optimal linear regression weights

2 Logistic Regression

The cross-entropy loss is defined as:

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

Where $\hat{y}(\mathbf{x}) = \sigma(W^T\mathbf{x} + b)$.

2.1 Loss Function and Gradient

Let's consider $\mathcal{L}_{\mathcal{D}} = -y \log \hat{y}(\mathbf{x}) - (1-y)\log(1-\hat{y}(\mathbf{x}))$. First, take derivatives of $\mathcal{L}_{\mathcal{D}}$ with respect to $\hat{y}(\mathbf{x})$:

$$\frac{d\mathcal{L}_{\mathcal{D}}}{d\hat{y}(\mathbf{x})} = -\frac{y}{\hat{y}(\mathbf{x})} + \frac{1-y}{1-\hat{y}(\mathbf{x})} \tag{1}$$

Then, take derivative of $\hat{y}(\mathbf{x})$ with respect to $\mathbf{x}$:

$$\frac{d\hat{y}(\mathbf{x})}{d\mathbf{x}} = \hat{y}(\mathbf{x})(1-\hat{y}(\mathbf{x})) \tag{2}$$

Next, take derivative of $\mathbf{x}$ with respect to weight:

$$\frac{\partial \mathbf{x}}{\partial W} = \mathbf{x} \tag{3}$$

Finally, combine equation (1), (2) and (3) and add regularization term, we have:

$$\frac{\partial \mathcal{L}_{\mathcal{D}}}{\partial W} = \frac{1}{N}\left[\mathbf{x}^T(\hat{y} - y)\right] + \sum_{n=1}^{N} 2\lambda w_i \tag{4}$$

Similarly,

$$\frac{\partial \mathcal{L}_{\mathcal{D}}}{\partial d} = \sum_{n=1}^{N} \frac{1}{N}(\hat{y} - y) \tag{5}$$

```
def crossEntropyLoss(W, b, x, y, reg):
    y_hat = 1.0/(1.0+np.exp(-(np.matmul(x,W)+b)))
    cross_entropy_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)
    print(cross_entropy_loss)
    return cross_entropy_loss
```

Listing 5: Cross Entropy Loss

```
def gradCE(W, b, x, y, reg):
    y_hat = 1.0/(1.0+np.exp(-(np.matmul(x,W)+b)))
    der_w =  np.matmul(np.transpose(x), (y_hat - y))/(np.shape(y)[0]) + 2*reg*W
    return der_w, np.sum((y_hat - y))/(np.shape(y)[0])
```

Listing 6: Cross Entropy Loss Gradient

2.2 Learning

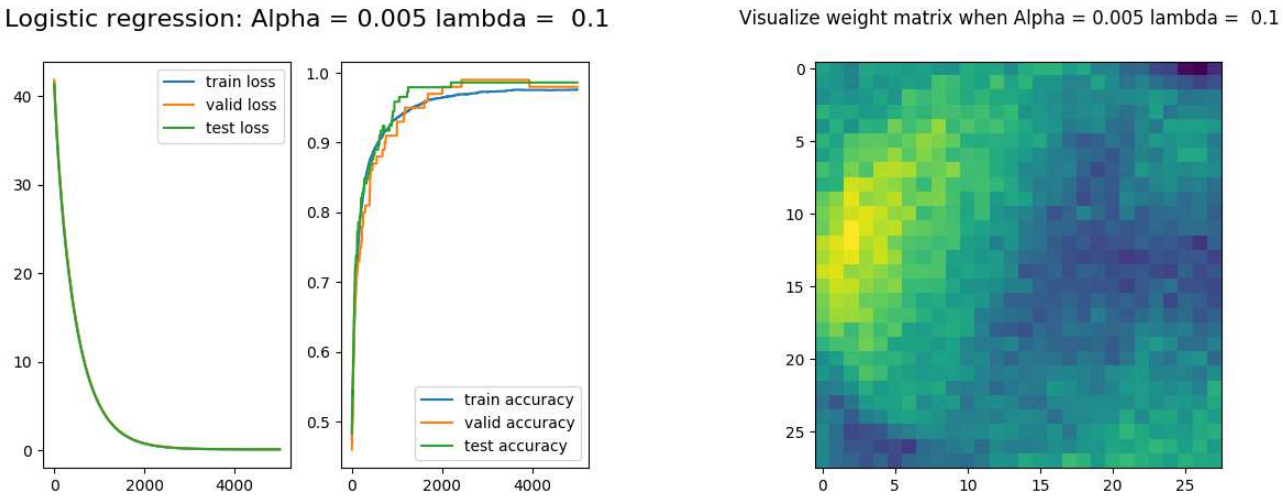

Figure 4: Logistic regression loss and accuracy curves and Visualization of Weight matrix

The gradient decent function that we had implemented with cross-entropy approach were similar to the gradient decent function by using MSE approach, the only difference was the replacement of MSE Loss function and gradMSE function with CrossEntropyLoss function and gradCE function.

The training model results were shown below in figure 4.

```
def grad_descent(W, b, trainingData, trainingLabels, alpha, iterations, reg, EPS):
    for i in range(iterations):
        grad_mse_W, grad_mse_b = gradCE(W, b, trainingData, trainingLabels, reg)
        new_W = W - alpha * grad_mse_W
        new_b = b - alpha * grad_mse_b
```

```
        mag = np.linalg.norm(new_W-W)
        if mag<EPS:
            return new_W,new_b
        else:
            W = new_W
            b = new_b
    return W,b
```

Listing 7: grad descent

2.3 Comparison to Linear Regression

Due to the reason of numerical stability, we had initialized weights by using Gaussian distribution with mean of 0, standard deviation of 0.5 for both MSE and CE.

Figure 5 shows the training loss for both MSE and CE approached models, for the figure, We could easily observe that logistic regression model using cross entropy loss converges much faster than linear regression model using MSE. This had indicated Cross-entropy is a batter measure than MSE for classification. This could be explained when deriving cost functions from the aspect of probability and distribution, MSE can be derived when assuming the error follows Normal Distribution, whereas the CE is based on the assumption of binomial distribution, this gives the natural that CE is better when doing classification (which is what we had done in this assignment), and MSE is better for doing regression.

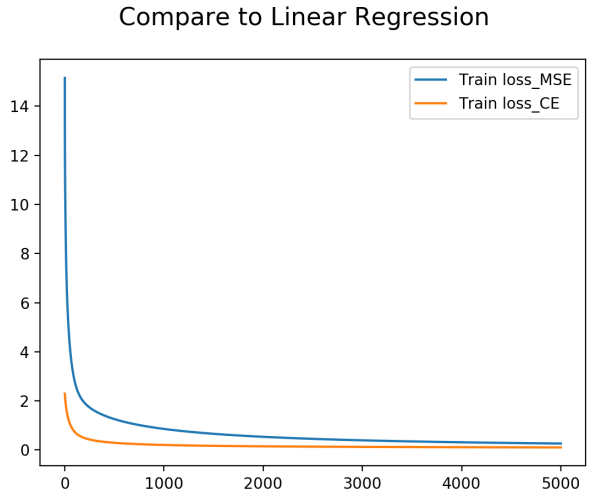

Figure 5: Training cross entropy loss and MSE loss for logistic regression and linear regression

3 Batch Gradient Descent vs. SGD and Adam

3.1 Building the Computational Graph

```
def buildGraph(beta1=None, beta2=None, epsilon=None, lossType=None, learning_rate=None):
    beta = 0
    graph = tf.Graph()
    batch_size = 500
    n_epochs = 700
    with graph.as_default():
        # Initialize weight and bias tensors
        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))

        x = tf.placeholder(tf.float32, shape=(batch_size, 784))
        y = tf.placeholder(tf.float32, shape=(batch_size, 1))
        # reg = tf.placeholder(tf.float32, shape = (1))

        valid_data = tf.placeholder(tf.float32, shape=(100, 784))
        valid_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))

        tf.set_random_seed(421)
        if lossType == "MSE":
            train_prediction = tf.matmul(x,W)+b
            loss = tf.losses.mean_squared_error(y, train_prediction)
            regularizer = tf.nn.l2_loss(W)
            loss = loss + beta/2.0 * regularizer

            optimizer = tf.train.AdamOptimizer(learning_rate=0.001, epsilon=1e-04).minimize(loss)
            # Predictions for the training, validation, and test data.


            valid_prediction = tf.matmul(valid_data,W)+b
            valid_loss = tf.losses.mean_squared_error(valid_label, valid_prediction)
            regularizer = tf.nn.l2_loss(W)
            valid_loss = valid_loss + beta/2.0 * regularizer

            test_prediction = tf.matmul(test_data,W)+b
            test_loss = tf.losses.mean_squared_error(test_label, test_prediction)
            regularizer = tf.nn.l2_loss(W)
            test_loss = test_loss + beta/2.0 * regularizer

        elif lossType == "CE":

            logits = tf.matmul(x, W) + b
            train_prediction = tf.sigmoid(logits)
            loss = tf.losses.sigmoid_cross_entropy(y, train_prediction)
            # Loss function using L2 Regularization
            regularizer = tf.nn.l2_loss(W)
            loss = loss + beta/2.0 * regularizer

            # Optimizer.
            optimizer = tf.train.AdamOptimizer(learning_rate=0.001, beta2=0.9999).minimize(loss)

            # 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)
            # Loss function using L2 Regularization
            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)
            # Loss function using L2 Regularization
            regularizer = tf.nn.l2_loss(W)
            test_loss = test_loss + beta/2.0 * regularizer

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

Listing 8: Build Graph

3.2 Implementing Stochastic Gradient Descent

```
def accuracy(predictions, labels):
    return (np.sum((predictions>=0.5)==labels) / np.shape(predictions)[0])

with tf.Session(graph=graph) as session:
    # This is a one-time operation which ensures the parameters get initialized as
    # we described in the graph: random weights for the matrix, zeros for the
    # biases.

    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]))
    # Run the computations. We tell .run() that we want to run the optimizer,
    # and get the loss value and the training predictions returned as numpy
    # arrays.
        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, train_prediction, 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))
            print('Loss at step {}: {}'.format(i, l))
            print('Training accuracy: {}'.format(accuracy(predictions, Y_batch)))

            # train_prediction
            print('Validation accuracy: {}'.format(accuracy(v_prediction, validTarget)))
            print('Test accuracy: {}'.format(accuracy(t_prediction, testTarget)))
```

Listing 9: Tensorflow Session

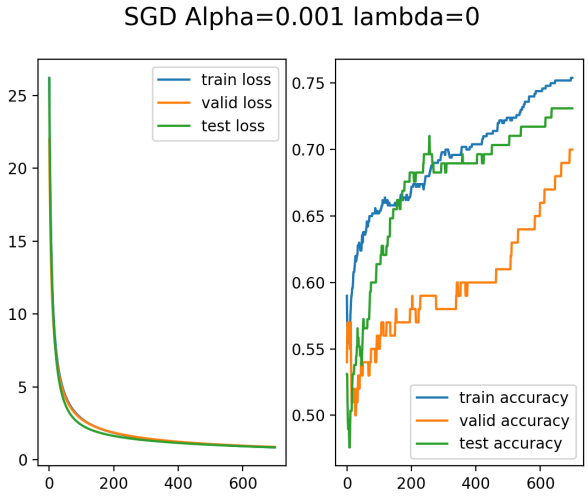

Figure 6: SGD for minibatch of 500 over 700 epochs

3.3 Batch Size Investigation

Figure 7 had shown the loss and accuracy changing curves for different batch sizes (100, 700, 1350). There were not much differences between loss curves for mini batch size 100 and 700, however, as the size of the mini batch increased to 1370, the graph showed a large increase on the loss and slower decay for all three data sets. The differences between different batch size could be observed more clearly on the accuracy curves. As the batch size increased, the models generated become less on the accuracy and consistency of training, valid, and test data sets. This is because mini batch has the

regularizing effects on the model in the way of adding extra noise to the gradient estimations. This helped the model to avoid bias estimations and also able to jerk the model out from local minimum, which led to better generalization. As the batch size increased, the noise become smaller, hence, there will be lose on the regularizing effect and lower the accuracy of validation and test data.

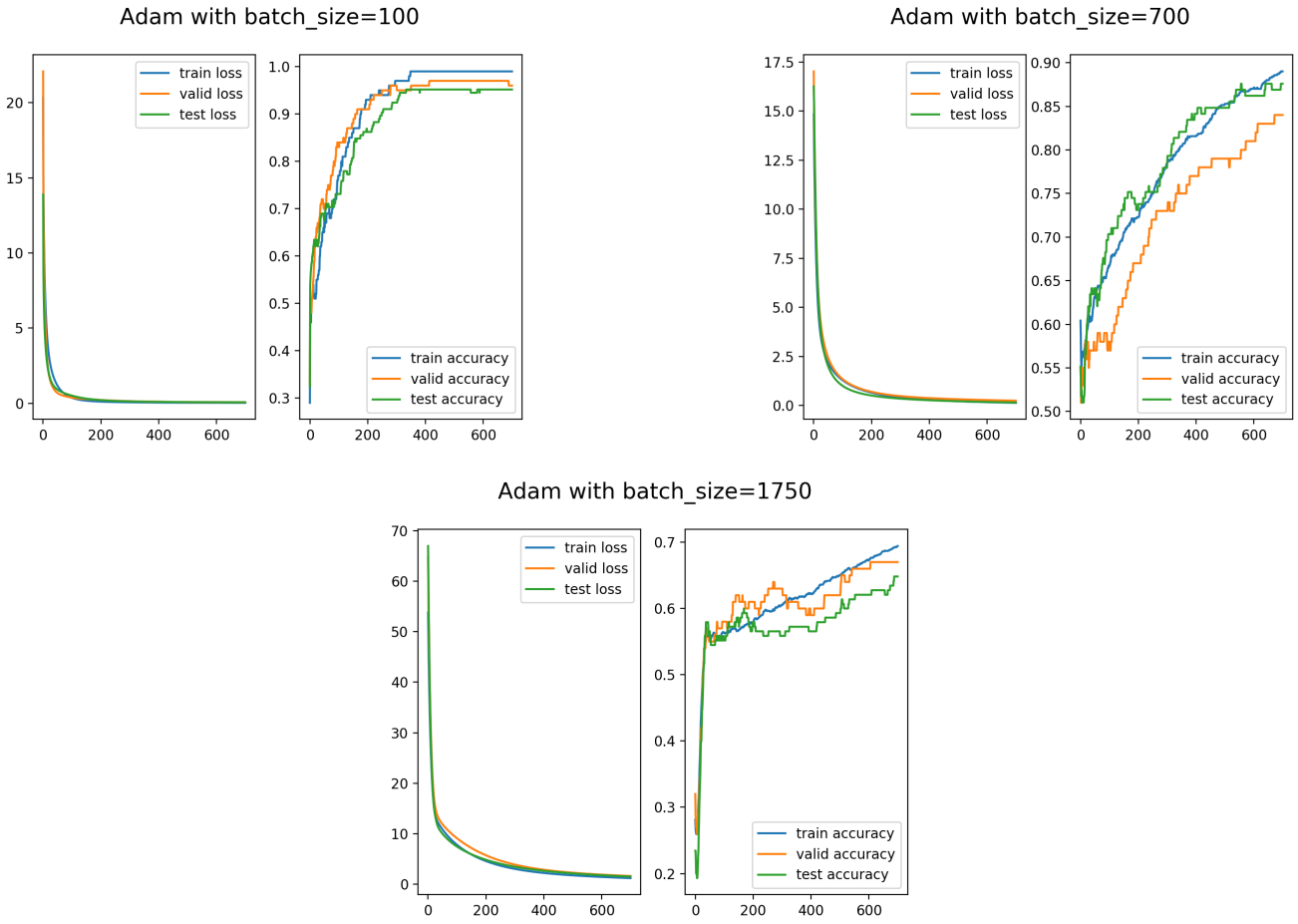

Figure 7: SGD optimized using Adam uising different batch size.

3.4 Hyperparameter Investigation

$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
  $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
  $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)

Figure 8: Adam hyper-parameters used for optimization. [1]

Figure 9 had shown the different accuracy curves' plots for train, valid, and test data in the context of changing Adam hyper-parameters β_1, β_2, and ϵ values. The default values are:

$$\beta_1 = 0.9 \qquad \beta_2 = 0.999 \qquad \epsilon = 10e - 08$$

	$\beta_1 = 0.95$	$\beta_1 = 0.99$	$\beta_2 = 0.99$	$\beta_2 = 0.9999$	$\epsilon = 1e-9$	$\epsilon = 1e-4$
Train data accuracy	91%	80%	93.8%	77.2%	89.2%	90.2%
Valid data accuracy	89%	79%	92%	79%	83%	84%
Test data accuracy	87.5%	82%	92.4%	81.4%	86.9%	84.1%

Table 4: Different Hyper-parameters final accuracy.

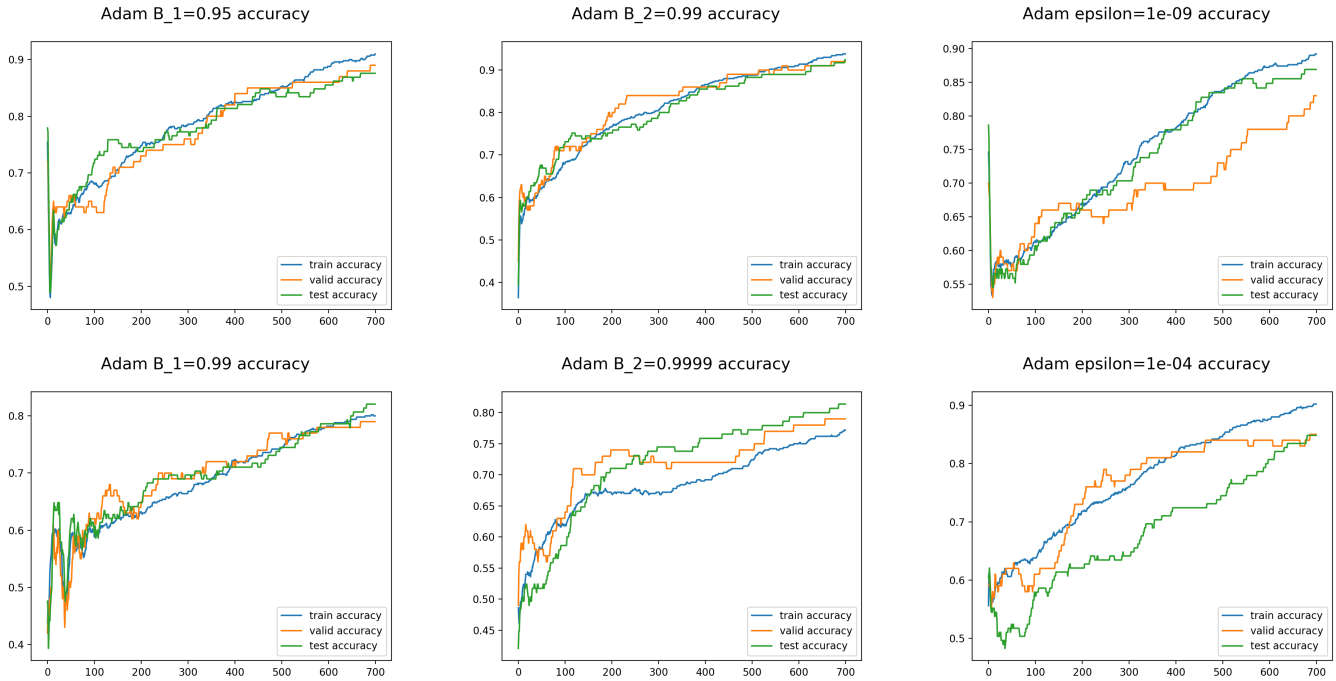

Figure 9: Adam hyper parameter impact on Accuracy

a) The Adam hyperparameter β_1 and β_2 are both use for speeding up the gradient decent. They are able to make the model converges faster than the standard gradient descent algorithm. β_1 is the exponential decay rate for the first moment estimates, which was adding momentum into consideration while doing gradient descent. By choosing a propriety value for β_1, each step size and direction were able to be adjusted, which led to faster convergent. first column on figure 9 had shown the accuracy curves plots with β_1 equals to 0.95 and 0.99. The 0.95 plot showed less oscillations at the beginning of the curves and reached a higher accuracy after 700 epochs. Since β_1 calculated current weight values by take historical values in to consideration. Higher the β_1 means more historical values had been took in for calculating the current weight values, therefore, when the β_1 equals 0.99, the model relaid heavily on the historical values (adding too much momentum), which gave less accuracy and more oscillations.

b) β_2 is the exponential decay rate for the second-moment estimates, it is similar as using Root Mean Square Propagation method when doing optimization. It works similar to the gradient descent algorithm for momentum (β_1), which could restrict the oscillation in the narrow direction and larger the steps in the wider direction. As shown in the figure 8 below, Vt is smaller when in the wider direction, and larger when in the narrow direction. Column 2 of Figure 9 had shown the accuracy curves for β_2 equals to 0.99 and 0.9999. As we can see, the accuracy was higher after 700 epochs with β_2 equals to 0.99 and lower when β_2 equals to 0.9999, and moreover, β_2 with value equals to 0.9999 was more inconsistent with train, valid, and test data accuracy curves. Is could also be explained by taken too much historical values while calculating new Vt, this had led to wrong direction taken for choosing the next weight.

c) ϵ value is a very small number to prevent a division by zero in the implementation. As shown in figure 8 above, the ϵ value was added to Vt in case Vt is equaling to zero. Column 3 of Figure 9 had shown the accuracy curves with ϵ equals to 1e-09 and 1e-04, as we could see, the over all accuracy for all three data sets are almost the same regardless of both ϵ values, however, curves for ϵ = 1e-09 has a big oscillation at the beginning of the curve, whereas the curves for ϵ = 1e-04 tend to be more smooth at the beginning. This met be caused by the zero initialize moment vectors, when ϵ = 1e-09 (very small number), there was less affect of ϵ towards weight parameters, therefore, there was oscillations at the beginning of the curves due to it had taken zero initialization into consideration. And when ϵ = 1e-04 (lager number), it

is large enough to affect the update weights parameters.

Notice, the accuracy plots may various due to data were randomly shuffled every time when training.

3.5 Cross Entropy Loss Investigation

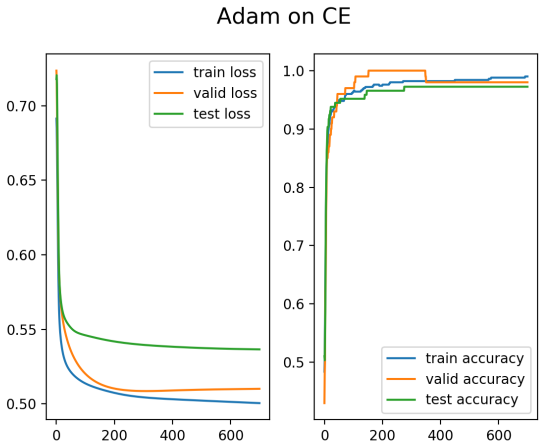

Figure 10: Minimizing binary cross entropy loss using Adam

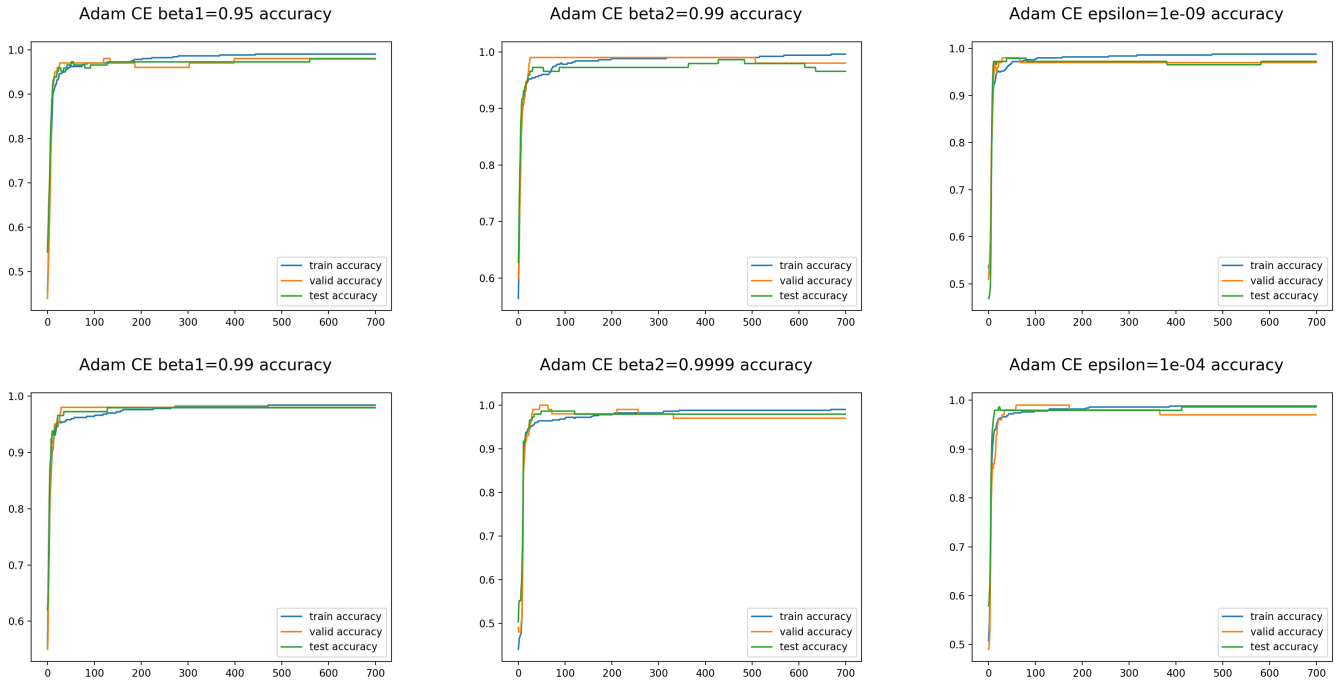

Figure 11: Adam hyper parameter impact on Accuracy on Cross Entropy

Figure 10 had shown the loss curves and accuracy curves for model using Adam cross entropy, compares to the model using MSE approach (Figure 6), the losses with CE model had starting at a much lower number (0.75 vs 25) and had faster decay. On the other side, for accuracy curves, instead of approximately 0.75 accuracy for MSE model, CE had reached close to one in fewer epochs. CE model also had higher consistency for the accuracy curves in all train, valid, and test data, which means it trained the model better than MSE. For changed β_1, β_2 or ϵ values on figure 10 for CE compared to figure 9 for MSE, CE approach had shown higher accuracy in every case compared to using MSE, and the affect of change in hyper-parameter for CE were very small. A possible reason for that was CE approach can reach a optimum training model with in 700 epochs much faster, therefore, less need for to speed up (optimization).

	$\beta_1 = 0.95$	$\beta_1 = 0.99$	$\beta_2 = 0.99$	$\beta_2 = 0.9999$	$\epsilon = 1e-9$	$\epsilon = 1e-4$
Train data accuracy	99%	98.4%	99.6%	99%	98.8%	98.8%
Valid data accuracy	98%	98%	98%	97%	97%	97%
Test data accuracy	98%	98%	96.6%	98%	97.2%	98.6%

Table 5: Different Hyper-parameters final accuracy.

3.6 Comparison against Batch GD

Due to smaller batch sizes and the Adam optimizer effect, the overall performance for training model using SGD algorithm with Adam is much better than using batch gradient decent algorithm that we had implemented in part 1 and 2. **In addition, for the loss plots:**

Batch GD algorithm:

- Loss for MSE (figure 2) was starting around 60 and reached almost zero after 5000 epochs.
- Loss for CE (figure 4) was starting around 40 and reached almost zero after 5000 epochs.

SGD algorithm with Adam:

- Loss for MSE (figure 6) was starting around 25 and reached almost zero after 700 epochs.
- Loss for CE (figure 10) was starting around 0.75 and reached almost zero after 700 epochs.

We could see SGD with Adam has much lower loss in each cases compared to use batch GD algorithm.

For the accuracy plots:

Batch GD algorithm:

- Accuracy for MSE (figure 2) was very inconsistent with three data sets and reached around 0.65 accurate after 5000 epochs.
- Accuracy for CE (figure 4) was consistent with three data sets and reached around 1.0 accurate after 5000 epochs.

SGD algorithm with Adam:

- Accuracy for MSE (figure 6) was very inconsistent with three data sets and reached around 0.75 accurate after 700 epochs.
- Accuracy for CE (figure 10) was consistent with three data sets and reached around 1.0 accurate after 700 epochs.

We could see SGD algorithm with Adam improved the final accuracy for MSE approach. For CE approach, there was not much difference between batch GD and SGD algorithms.

There are several possible explanations of why SGD with Adam gives a general better performance in perspectives of speed, loss, and accuracy. firstly, by using smaller sized batches, the calculation time will be much faster compared to use the whole data set to compute. Secondly, by using the Adam optimizer, it could lead to a faster convergence rate by using a more accurate step direction towards minimum value. Thirdly, Adam optimizer can also adds momentum when determining Weights, which helped the model to escape shallow local minimums.

References

[1] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv e-prints*, p. arXiv:1412.6980, Dec. 2014.