ECE421 Fall 2019 Assignment 1 - Linear and Logistic Regression

Name: Anirudh Sampath (50%), Anas Akram Musharraf (50%)
SN: 1002630218, 1002428072

NOTE: The following is a quick overview of the notation that was used throughout this assignment:

- x: Data Matrix
  - Dimensions: [nxd]
  - n is defined as the number of images in the data set
  - d is defined as the flattened dimension of the image (for nMIST d = 28x28=784)
- W: weight matrix
  - Dimensions: [dx1] where d is as defined above
- y: Data Label
  - Dimensions [nx1] where n is defined as above
- b: Bias
  - dimensions: [nx1] where n is defined as above
  - **NOTE:** In numpy, b was defined as a single number but due to the nature of the np.sum operation, b was treated as a [nx1] vector
- reg: λ or regularization parameter
  - dimensions: [1x1] constant value
- reg: α or learning rate
  - dimensions: [1x1] constant value
- epochs: Number of full cycles through the data set
  - dimensions: [1x1] constant value
- error-tol: error tolerance (termination condition for gradient descent)
  - dimensions: [1x1] constant value

1. Linear Regression

1.1 Loss function and Gradient

Below are the analytical expressions for the mean-squared-error (MSE) and gradient of the mean-squared-error (gradMSE), and their respective numpy functions:

Mean Squared Error:
$MSE = \frac{1}{N}\sum_{n=1}^{N}||x_nW + b - y_n||_2^2 + \frac{\lambda}{2}||W||_2^2$

Below is the numpy function for MSE:

```
def MSE(W, b, x, y, reg):
    constant = y.shape[0]
    error = np.matmul(x,W)
    error_2 = error + b - y
    term1 = np.sum(error_2*error_2 , axis=0)
    term2 = (reg/2)*np.sum(W*W)
    return (1/constant)*term1 + term2
```

Gradient of Mean Squared Error with respect to W:
This was calculated by taking the partial derivative of the mean squared error with respect to the weight vector. The analytical expression is shown below, and the steps taken have been attached in the appendix.

$gradMSE_W = \frac{2}{N}\sum_{n=1}^{N}||x_nW + b - y_n||x_n + \lambda||W||$

Gradient of Mean Squared Error with respect to b:
This was calculated by taking the partial derivative of the mean squared error with respect to the bias. The analytical expression is shown below, and the steps taken have been attached in the appendix.

$gradMSE_b = \frac{2}{N}\sum_{n=1}^{N}||x_nW + b - y_n||$

Below is the numpy function for gradMSE, which returns gradient with respect to both W and b:

```
def gradMSE(W, b, x, y, reg):
    constant = y.shape[0]
    error = np.matmul(x,W)
    error_2 = error + b - y
    term1 = (error_2)*x
    term1 = np.sum(term1, axis=0).reshape((W.shape[0],1))
    grad_wrt_w = (2/constant)*term1 + reg*W
    grad_wrt_bias = (2/constant)*np.sum(np.matmul(x,W)+b-y, axis=0)
    return grad_wrt_w, grad_wrt_bias
```

Q1.2 Gradient Descent

Gradient descent is one of many strategies that computer scientists use to train algorithms to accurately classify data points. The aim of gradient descent is to minimize the error function, and it does so using iteration. In essence, we initialize a weight and bias vector, and use the gradients of the error function with respect to these variables, to adjust the weight and bias vectors. We repeat this until the weight and bias vectors start to converge, or if we reach the maximum number of iterations.

Below is the numpy function for gradientDescent, which returns gradient with respect the updated weight and bias vectors:

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol):
    # Your implementation here
    current_weight =W
    current_bias = b
    for i in range(epochs):
        grad_weight, grad_bias = gradMSE(current_weight,current_bias,x,y,reg)
        updated_weight = current_weight - alpha*grad_weight
        updated_bias = current_bias - alpha*grad_bias
        if alpha*np.sum(grad_weight*grad_weight,axis=0) < error_tol:
            return current_weight, current_bias
        current_bias,current_weight = updated_bias, updated_weight
    return updated_weight, updated_bias
```

Notes on Gradient Descent Implementation:

- **Update Rule**:
  - $W \leftarrow W - \alpha\frac{\delta Error}{\delta W}$
  - $b \leftarrow b - \alpha\frac{\delta Error}{\delta b}$
- The α is defined as the learning rate, and is a parameter that controls the size of the correction of the weight and bias vectors during each iteration. As α increases, the step size of the correction factor increases as well.
- **Early termination condition**:
  - $\alpha||\frac{\delta Error}{\delta W}||^2 < error_tolerance$
  - The early termination condition for the gradient descent algorithm is when the difference between consecutive weight vector iterations is very small. This is a sign of the algorithm slowing down or a sign of the algorithm converging around the optimal weight vector.

Q1.3 Tuning the learning rate

After defining the gradDescent function, we trained the algorithm over 5000 epochs and used a regularization parameter (λ) of 0. We tracked the losses of the gradient descent algorithm after running through the training, validation and test data while modulating the learning rate. The loss was calculated by feeding in the appropriate inputs into the mean squared error function written in **Q1.1**. The data is presented in the following table.

Note: The entries represent the loss values after 5000 epochs:

	$\alpha = 0.005$	$\alpha = 0.001$	$\alpha = 0.0001$
Training loss	0.02541	0.03514	0.05897
Valid loss	0.03012	0.03722	0.06818
Test loss	0.03527	0.03949	0.06249

Table 1: Effect of modulating the learning rate (α) on training, validation and test loss after 5000 epochs.

Below are the plots showing how the training, validation and test losses are affected when changing the learning rate (α).

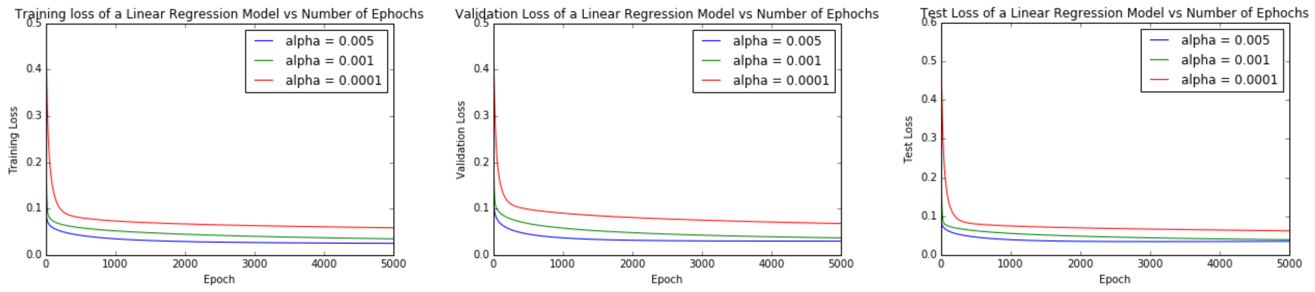


Figure 1: Training, validation and test losses at different values of α plotted against number of epochs.

Comments:

- Training, validation and test losses decreases most significantly during the first 500 epochs when running the gradient descent algorithm with the mean squared error loss function
- Increasing the learning rate (α) results in significantly lower losses for all three data sets.
- Algorithm appears to converge faster as the learning rate is increased (higher initial negative slope)
- Thus, we can confirm that the algorithm is behaving as expected

Q1.4 Generalization of gradDescent algorithm

The following section is an investigation into the effect of modulating the regularization parameter on the accuracy of the algorithm with respect to the training, validation and test data sets. The formula used to calculate the accuracy as well as the python snippet is shown below:

$$Accuracy = \frac{1}{N}\sum_{n=1}^{N}(\hat{y} - y)$$

Since, the weight and bias vectors are generated through gradient descent, we need to round the $\hat{y}$ (predicted labels) appropriately to ensure that we can maintain the binary nature of the data set. The numpy function used to calculate accuracy for MSE and Cross-entropy is shown below:

```
def calculated_accuracy(W,b,x,y, losstype="MSE"):
    if losstype=="CE":
        updated_prediction = sigmoid(W,x,b)
    else:
        updated_prediction = np.matmul(x,W) + b
    updated_prediction = np.round(updated_prediction)
    accuracy = np.sum(updated_prediction==y)/np.shape(x)[0]
    return accuracy
```

Note: The entries represent the accuracy values after 5000 epochs:

	$\lambda = 0.001$	$\lambda = 0.1$	$\lambda = 0.5$
Training accuracy	0.98457	0.98257	0.982
Valid accuracy	0.98	0.99	0.99
Test accuracy	0.97931	0.97931	0.98621

Table 2: Effect of modulating the learning rate (λ) on training, validation and test accuracy after 5000 epochs.

Below are the plots showing how the training, validation and test accuracy are affected by the regularization parameter (λ).

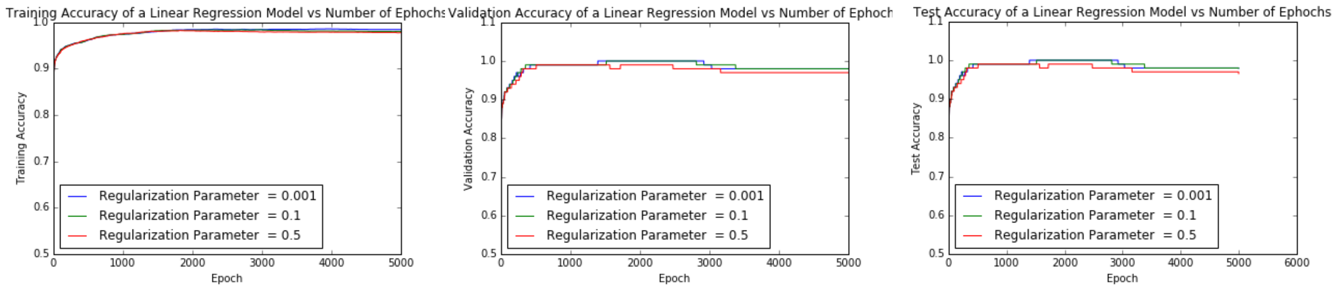


Figure 2: Training, validation and test accuracy at different values of λ plotted against number of epochs.

Comments:

- It appears that modulating λ has very minimal impact on the accuracy and similarly does not affect the speed of convergence (all 3 values of λ showed nearly identical graphs)
- The graphs for the validation and test accuracy are not smooth like the training accuracy graph. This can be attributed to the fact that there are significantly more data points in the training set, and that the bias and weight vectors generated are tuned to the training set.
- The validation and test accuracy appears to reach a maximum at around 2000 epochs and starts to decrease very slightly after that. This can be an example of overfitting, where the weight and bias vectors are very finely tuned to the training set.

1.5 Batch gradient descent vs Normal Equation

The normal equation is the closed form analytical solution to the minimization of the gradient of the mean squared error. Assuming zero weight decay (λ =0), the following is the closed form equation for used to obtain the optimal weight vector:

$$W = (X^T X)^{-1} Y$$

Below is the numpy function used:

```
def normal_equation(x, y):
    d = np.shape(x)[0]
    one_vec = np.expand_dims(np.ones(d),1)
    x_aug = np.concatenate((x,one_vec),axis = 1)
    A = np.linalg.inv(np.matmul(np.transpose(x_aug),x_aug))
    B = np.matmul(np.transpose(x_aug),y)
    W_aug = np.matmul(A,B)
    W = W_aug[:len(W_aug)-1]
    b = (W_aug[len(W_aug) -1]).reshape([1,1])
    return W,b
```

NOTE: Gradient descent in the table refers to batch gradient descent with λ = 0, 5000 epochs and $\alpha = 0.005$

	Gradient Descent	Normal Equation
Training loss	0.307	0.0187
Validation loss	0.4058	0.04756
Test loss	0.3533	0.056979
Training accuracy	0.684	0.993714
Valid accuracy	0.57	0.96
Test accuracy	0.655	0.9379
Computation time	22s	0.111s

Table 3: Comparison of loss and accuracy when using gradient descent vs normal equation.

From our testing, the results with the normal equation are significantly better than those obtained through this version of gradient descent. We are able to more finely tune the weight and bias vectors with the normal equation than through adjusting the gradient of the mean squared error. Even in terms of computation time, the normal equation was computed significantly faster than through gradient descent. It seems that when modelling with λ =0, the gradient descent algorithm does not perform well at all, and that even small levels of weight decay significantly alter the performance of the algorithm.

2. Logistic Regression

2.1 Cross-entropy Loss function and Gradient

Below are the analytical expressions for cross-entropy loss (CE) and gradient of the cross-entropy loss (gradCE), and their respective numpy functions:

Cross-entropy loss function:

$CE = \sum_{n=1}^{N} \frac{1}{N} \Bigg[- y^{(n)} \log \hat{y}(\mathbf{x}^{(n)}) - (1 - y^{(n)}) \log(1 - \hat{y}(\mathbf{x}^{(n)})) \Bigg] + \frac{\lambda}{2} \|\mathbf{W}\|_2^2$

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

Below is the numpy function for CE:

```
def crossEntropyLoss(W, b, x, y, reg):
    # Your implementation here
    a = y * np.log(sigmoid(W,x,b))
    b = (1 - y) * np.log(1 - sigmoid(W,x,b))
    N = y.shape[0]
    term1 = np.sum(-a - b, axis=0)/N
    term2 = (reg/2)*np.sum(W*W)
    return term1 + term2
```

Gradient of Cross Entropy Loss with respect to W:
This was calculated by taking the partial derivative of the cross entropy loss with respect to the weight vector. The analytical expression is shown below, and the steps taken have been attached in the appendix.

$gradCE_W = \frac{1}{N} \sum_{n=1}^{N} (\hat{y_n} - y)x_n + \lambda W$
$gradCE_W = \frac{1}{N} \sum_{n=1}^{N} (\sigma(x_n W + b) - y)x_n + \lambda W$

Gradient of Cross Entropy Loss with respect to b:
This was calculated by taking the partial derivative of the cross entropy loss with respect to the bias. The analytical expression is shown below, and the steps taken have been attached in the appendix.

$gradCE_b = \frac{1}{N} \sum_{n=1}^{N} (\hat{y_n} - y)$
$gradCE_b = \frac{1}{N} \sum_{n=1}^{N} (\sigma(x_n W + b) - y)$

Below is the numpy function for gradCE, which returns gradient with respect to both W and b:

```
def gradCE(W, b, x, y, reg):
    # Your implementation here
    N = y.shape[0]
    y_hat = sigmoid(W,x,b)
    y_hat_min_y = y_hat-y
    part1Done = y_hat_min_y*x
    sol = (1/N)*np.sum(part1Done, axis=0).reshape([W.shape[0],1]) + reg*W
    part2 = (1/N)*np.sum(y_hat_min_y, axis=0)
    return sol, part2
```

2.2 Using logistic regression in gradient descent

Below is the updated gradient descent algorithm to allow for usage of the cross-entropy loss and gradient:

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol, lossType="MSE"):
    # Your implementation here
    current_weight = W
    current_bias = b
    for i in range(epochs):
        if lossType == "MSE":
            grad_weight, grad_bias = gradMSE(current_weight,current_bias,x,y,reg)
        elif lossType=="CE":
            grad_weight, grad_bias = gradCE(current_weight,current_bias,x,y,reg)
        updated_weight = current_weight - alpha*grad_weight
        updated_bias = current_bias - alpha*grad_bias
        if alpha*np.sum(grad_weight*grad_weight,axis=0) < error_tol:
            return current_weight, current_bias
        current_bias,current_weight = updated_bias, updated_weight
    return updated_weight, updated_bias
```

The following is an investigation into the performance of the gradient descent algorithm when applying logistic regression. The model was trained over 5000 epochs, after choosing a learning rate of 0.005 and a regularization parameter value of 0.1. The training, validation and test losses and accuracy are shown in the plots below:

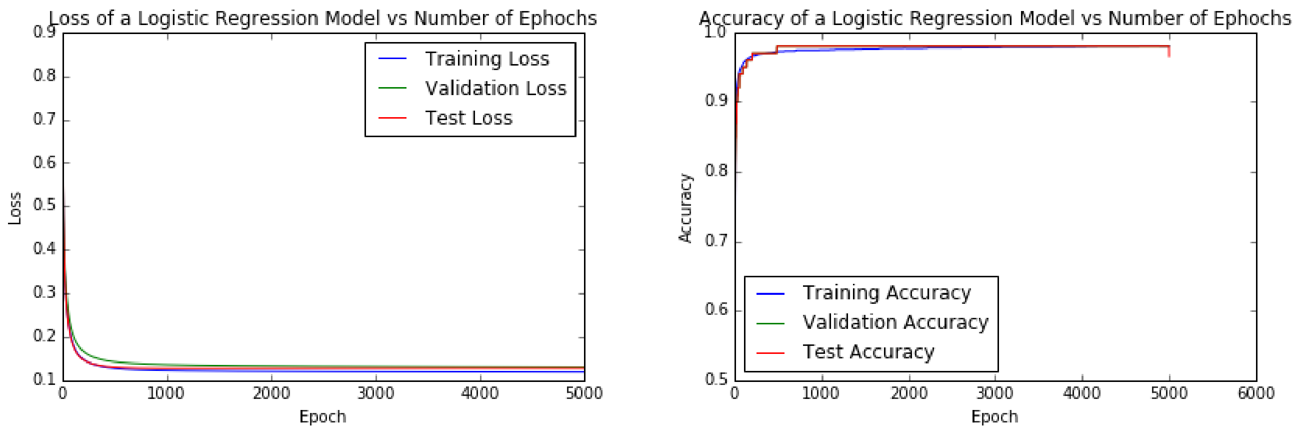


Figure 3: Training, validation and test accuracy at α=0.005 and λ=0.1 vs number of epochs.

Comments:

- The initial slope of the loss is quite steep, and the loss plateaus after about 1000 epochs. This shows that the algorithm is able to converge to optimal weight and bias vectors rather quickly.
- The shapes of the training, validation and test accuracy curves are rather similar, but the graphs for the validation and test accuracy are not smooth like the training accuracy graph. This can be attributed to the fact that there are significantly more data points in the training set, and that the bias and weight vectors generated are tuned to the training set.

2.3 Comparing Logistic and Linear Regression

Below is a comparison of the training loss calculated when using mean squared error and cross-entropy loss:

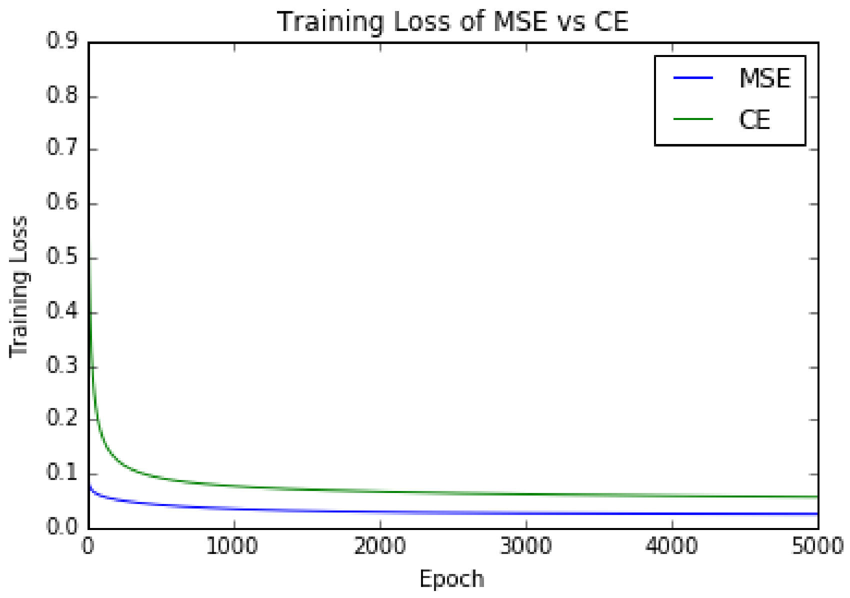


Figure 4: Training calculated when using mean squared error vs cross-entropy loss.

This difference in training loss calculated when running linear and logistic regression is primarily a result of the difference in function used for error. Linear regression uses a loss function that is a squared relation of the difference between the predicted values and the data labels. Logistic regression uses the logarithmic loss function, as well as the sigmoid function, which results in an exponential relation. The key difference between the two error functions is that the log loss function is more sensitive to ”very correct” or ”very incorrect” data points. As a result, the cross entropy loss converges at a high value than the mean squared error.

3. Batch Gradient Descent vs Stochastic Gradient Descent and ADAM

3.1 Building the tensorflow computational graph

Below is the buildGraph() function that we used to build the tensorflow computational graph.

NOTE: The buildGraph() function has been updated to allow for changing of various parameters as well as allowing for data inputs other than just the training set. Also, the tensorflow session has been initialized within this function and thus, the implementation of stochastic gradient descent is located within this function.

```
def buildGraph(loss="MSE", betaOne="None", betaTwo="None", epsilon="None"):
    lambd = tf.constant([0.0])
    #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=(None, 784))
    y = tf.placeholder(tf.float32, shape=(None, 1))
    reg = tf.constant([0.0])

    tf.set_random_seed(421)

    if loss == "MSE":
        y_Pred = tf.matmul(x,W) + b
        loss = tf.losses.mean_squared_error(y,y_Pred)
        loss = loss + lambd * reg
        optimizer = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)
        y_Pred = tf.round(y_Pred)


    elif loss == "CE":
        sigmoid_term = tf.matmul(x,W) + b
        y_Pred = tf.sigmoid(sigmoid_term)
        loss = tf.losses.sigmoid_cross_entropy(y, sigmoid_term)
        loss = loss + lambd * reg
        optimizer = tf.train.AdamOptimizer(learning_rate=0.001,epsilon=epsilon).minimize(loss)
        y_Pred = tf.round(y_Pred)


    trainingLoss = []
    trainingAcc = []
    validationLoss = []
    validationAcc = []
    testingLoss = []
    testingAcc = []
    init = tf.global_variables_initializer()
    with tf.Session() as session:

        session.run(init)
        trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
        n_batches = int(3500/batch_size)
        for i in range(epochs):
            #Shuffle here

            trainData_shaped = reshape_data_tensor(trainData)
            random_order = np.random.permutation(trainData_shaped.shape[0])
            trainData_random = trainData_shaped[random_order]
            trainTarget_random = trainTarget[random_order]


            validData_shaped = reshape_data_tensor(validData)
            random_order = np.random.permutation(validData_shaped.shape[0])
            validData_random = validData_shaped[random_order]
            validTarget_random = validTarget[random_order]


            testData_shaped = reshape_data_tensor(testData)
            random_order = np.random.permutation(testData_shaped.shape[0])
            testData_random = testData_shaped[random_order]
            testTarget_random = testTarget[random_order]


            training_loss_in_batch = 0
            valid_loss_in_batch = 0
            testing_loss_in_batch = 0

            trainingPredictions = []
            validPredictions = []
            testingPredictions = []

            trainingCorrect_points = 0
            validCorrect_points = 0
            testCorrect_points = 0
            for i in range(n_batches):
                #print(i)
                #pick first 500 from training data
                feed_x = trainData_random[i*batch_size:((i+1)*batch_size)]
                feed_y = trainTarget_random[i*batch_size:((i+1)*batch_size)]

                validFeed_x = validData_random[i*batch_size:((i+1)*batch_size)]
                validFeed_y = validTarget_random[i*batch_size:((i+1)*batch_size)]

                testingFeed_x = testData_random[i*batch_size:((i+1)*batch_size)]
                testingFeed_y = testTarget_random[i*batch_size:((i+1)*batch_size)]


                session.run(optimizer, feed_dict={x:feed_x, y:feed_y})
                training_loss_in_batch += session.run(loss, feed_dict={x: feed_x, y:feed_y})[0]
                valid_loss_in_batch += session.run(loss, feed_dict={x: validFeed_x, y:validFeed_y})[0]
                testing_loss_in_batch += session.run(loss, feed_dict={x: testingFeed_x, y:testingFeed_y})[0]


                trainingPredictions = session.run(y_Pred,feed_dict={x: feed_x, y:feed_y})
                trainingPredictions = np.asarray(trainingPredictions)
                validPredictions = session.run(y_Pred,feed_dict={x: validFeed_x, y:validFeed_y})
                validPredictions = np.asarray(validPredictions)
                testingPredictions = session.run(y_Pred,feed_dict={x: testingFeed_x, y:testingFeed_y})
                testingPredictions = np.asarray(testingPredictions)

                trainingCorrect_points += np.sum(trainingPredictions==feed_y)
                validCorrect_points += np.sum(validPredictions==validFeed_y)
                testCorrect_points += np.sum(testingPredictions==testingFeed_y)

            trainingPredictions = np.asarray(trainingPredictions)
            validPredictions = np.asarray(validPredictions)
            testingPredictions = np.asarray(testingPredictions)


            trainingAcc.append(trainingCorrect_points/trainTarget_random.shape[0])
            trainingLoss.append(training_loss_in_batch/n_batches)
            validationAcc.append(validCorrect_points/validTarget_random.shape[0])
            validationLoss.append(valid_loss_in_batch/n_batches)
            testingAcc.append(testCorrect_points/testTarget_random.shape[0])
            testingLoss.append(testing_loss_in_batch/n_batches)
            #print(trainingAcc)

    return W,b,y_Pred,y,loss,optimizer, trainingLoss, trainingAcc,validationLoss,validationAcc, testingAcc,testingLoss
```

3.2 Implementation of Stochastic Gradient Descent

NOTE: The python implementation of stochastic gradient descent is in the buildGraph() function of 3.1.

Below are the plots for the training, validation and test losses and accuracy. The regularization parameter (λ) was set to 0 and the learning rate (α) was set to 0.001. The mean squared error loss functions was used in the training of this model.

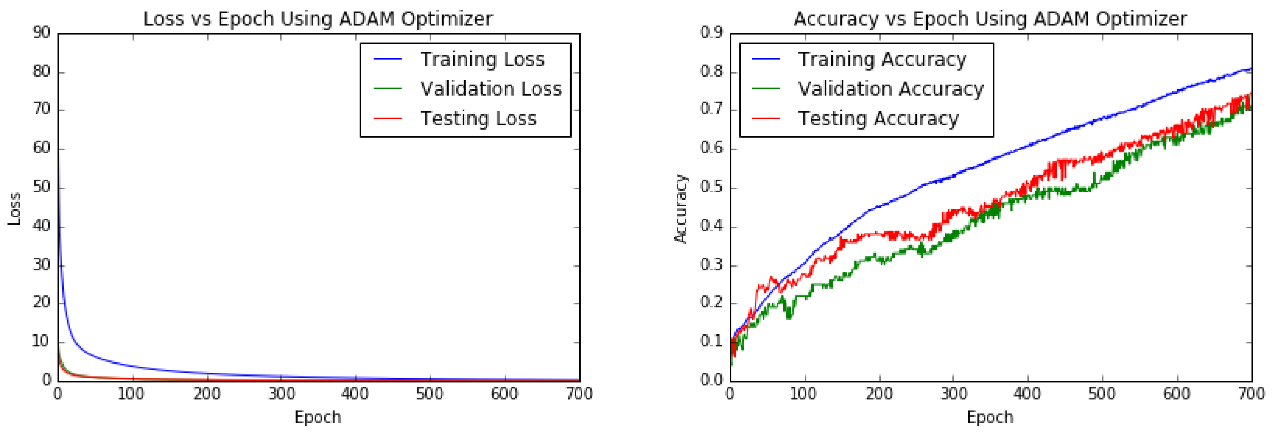


Figure 5: Training, validation and test loss and accuracy using the ADAM optimizer and stochastic gradient descent at α=0.001 and λ=0 vs number of epochs.

Comments:

- We see how using the ADAM optimizer and SGD our loss converges to zero much faster than previous methods.
- But using the ADAM optimizer and SGD we do not achieve great accuracy even for the training for which we have a lot of data.

3.3 Investigation of impact of batch size on loss and accuracy

Below are the plots of training, validation, and test loss and accuracy at different batch sizes:

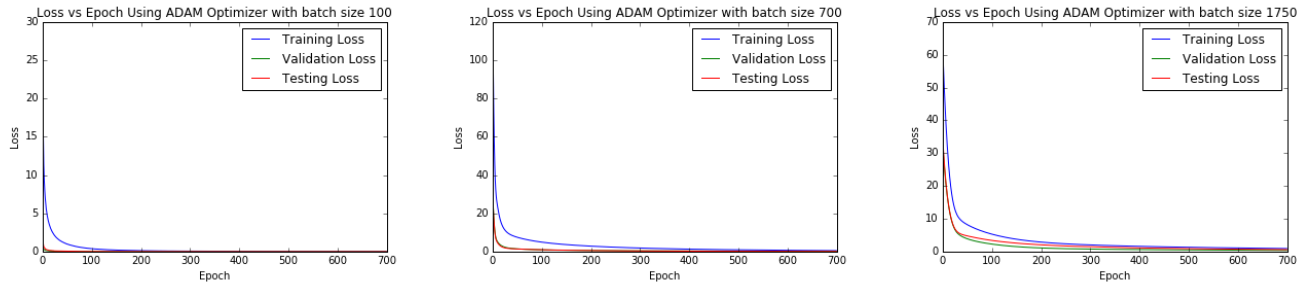


Figure 6: Training, validation and test loss using the ADAM optimizer and stochastic gradient descent at α=0.001, λ=0 and batch sizes of 100, 700, 1750 vs number of epochs.

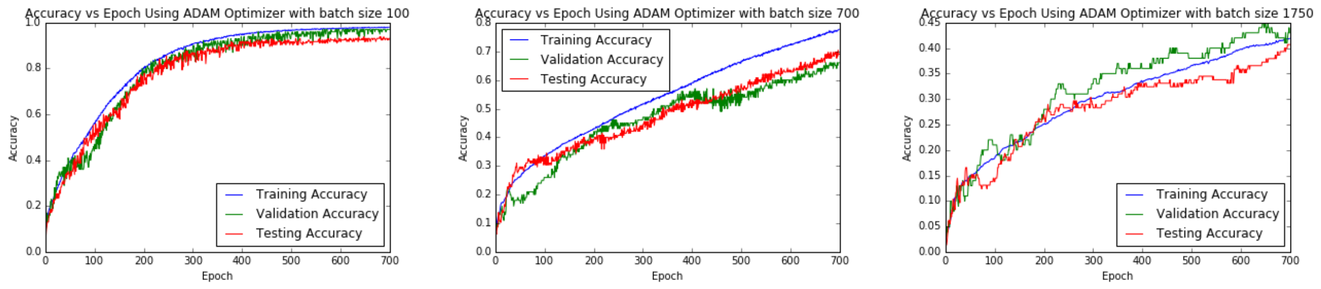


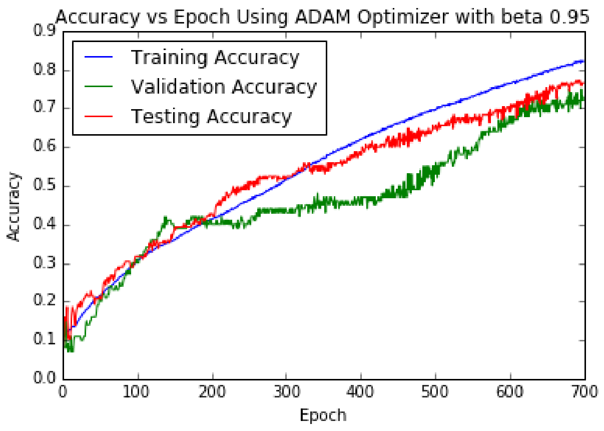
Figure 7: Training, validation and test accuracy using the ADAM optimizer and stochastic gradient descent at α=0.001, λ=0 and batch sizes of 100, 700, 1750 vs number of epochs.

Comments:

- First thing to note is how the smaller the batch size, the faster our loss converges to zero and the higher our accuracy. We expect this because a lower batch size means we have more batches when going through the data. This allows us to develop a better weight vectors to apply on the validation and test data.
- Even though we can clearly see the difference in the losses for the different batch sizes, the impact of batch sizes is even more prevalent when comparing the accuracy. It is important to note how when the batch size is 100, we achieve nearly 100 percent accuracy with each graph being relativly smooth. Whereas with larger batch sizes, for example with batch sizes of 1750, we achieve an accuracy of 45 percent. This is what we expect because with larger batch sizes we do not iterate through the data as much, this prevents us from developing a highly optimized weight vector and this is shown in the results.
- With a smaller batch size, we perform SGD more times during each epoch. As we are plotting the loss and accuracy against the number of epochs, and not the total number of iterations of SGD, it is expected that we see better results at lower batch size.

3.4.A Investigation of impact of hyperparamaters on ADAM optimizer - β_1

Below are the plots of training, validation, and test accuracy and the effect of modulating hyperparameter β_1 :



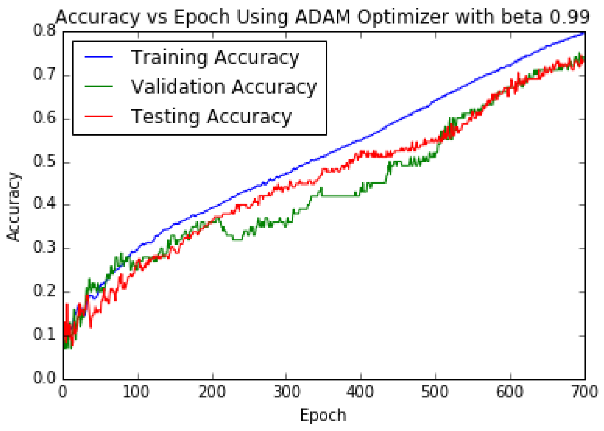


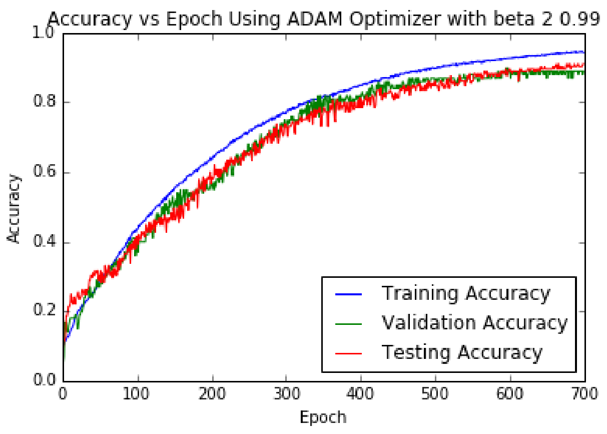
Figure 8: Training, validation and test loss using the ADAM optimizer and stochastic gradient descent at α=0.001, λ=0 and modulating β_1 value vs number of epochs.

Comments:

- ADAM optimizer calculates the exponential moving average of the gradient and squared gradient. β_1 signifies the exponential decay of the first gradient.
- β_1 does not change the shape of the graphs too much but when β_1 was changed from 0.95 to 0.99 we saw the accuracy drop around 10 percent.
- β_1 affects the rate at which the learning rate decays. Thus, increasing the beta one value the caused the accuracy's to decline because the learning rate varied more between batches and epochs.

3.4.B Investigation of impact of hyperparamaters on ADAM optimizer - β_2

Below are the plots of training, validation, and test accuracy and the effect of modulating hyperparameter β_2 :



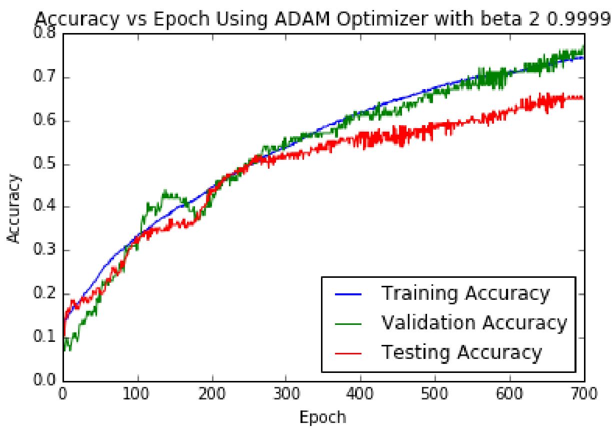


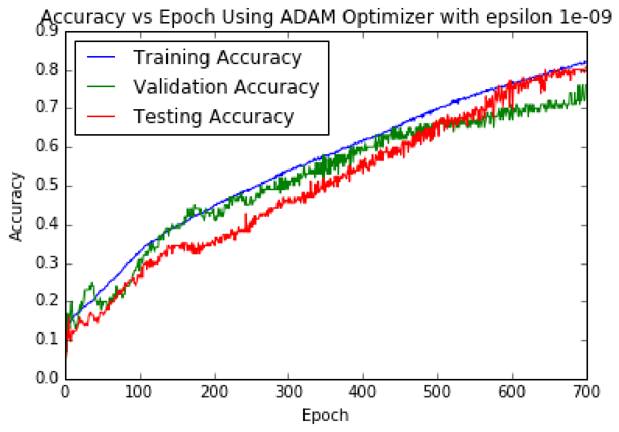
Figure 9: Training, validation and test loss using the ADAM optimizer and stochastic gradient descent at α=0.001, λ=0 and modulating β_2 value vs number of epochs.

Comments:

- ADAM optimizer calculates the exponential moving average of the gradient and squared gradient. β_2 signifies the exponential decay of the squared gradient.
- Firstly we notice how using a larger β_2 causes there to be lower accuracies and more instability. This is caused because with higher β_2 we increased the rate at which learning rate decays, and this makes our momentum goes to zero faster and that hinders our performance.

3.4.C Investigation of impact of hyperparamaters on ADAM optimizer - ϵ

Below are the plots of training, validation, and test accuracy and the effect of modulating hyperparameter ϵ :



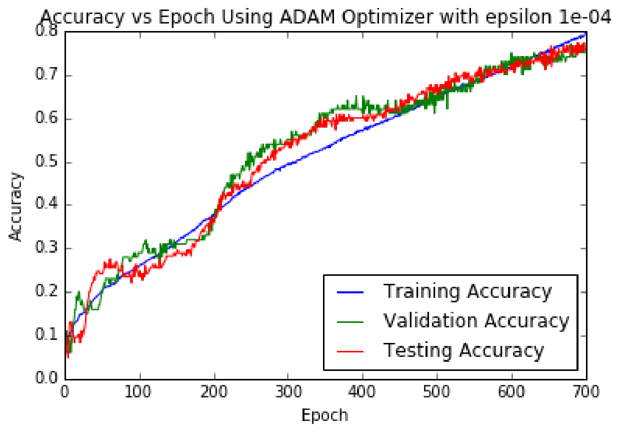


Figure 10: Training, validation and test loss using the ADAM optimizer and stochastic gradient descent at α=0.001, λ=0 and modulating ϵ value vs number of epochs.

Comments:

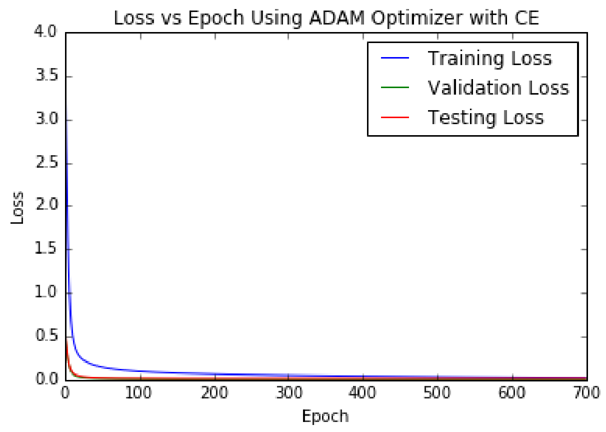
- The ϵ term exists in the ADAM optimizer so that it does not encounter a situation where it has to divide by zero when the gradient is almost zero.
- The change in ϵ did not have a significant impact on our accuracies and this this is expected because it does not effect something like our learning rate like beta one and beta two.

3.5.1 - Included as part of 3.1

3.5.2 Implementation of Stochastic Gradient Descent - Cross Entropy Loss

NOTE: The python implementation of stochastic gradient descent is in the buildGraph() function of 3.1.

Below are the plots for the training, validation and test losses and accuracy. The regularization parameter (λ) was set to 0 and the learning rate (α) was set to 0.001. The cross entropy loss functions was used in the training of this model.



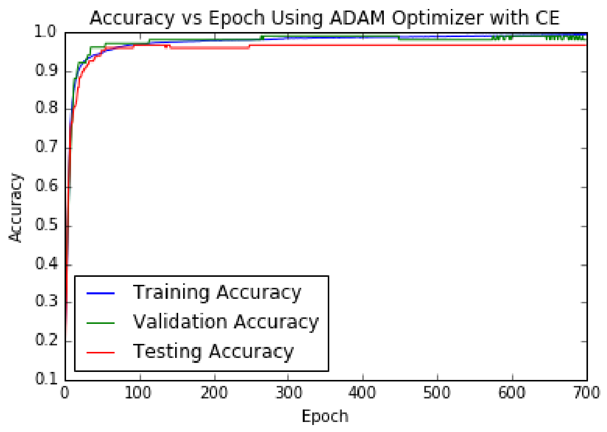


Figure 11: Cross-entropy Training, validation and test loss and accuracy using the ADAM optimizer and stochastic gradient descent at α=0.001 and λ=0 vs number of epochs.

Comments:

- We can easily see how using cross entropy with the ADAM optimizer produces better results than MSE. The losses converge to zero much quicker and the accuracies are all stable and nearly 100 percent.

3.5.3 Investigation of impact of batch size on loss and accuracy - Cross Entropy Loss

Below are the plots of training, validation, and test loss and accuracy at different batch sizes:

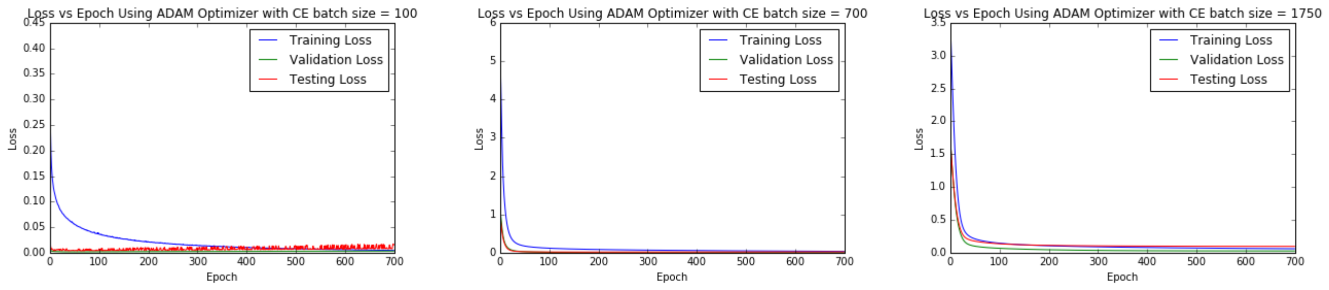


Figure 12: Cross-entropy training, validation and test loss using the ADAM optimizer and stochastic gradient descent at α=0.001, λ=0 and batch sizes of 100, 700, 1750 vs number of epochs.

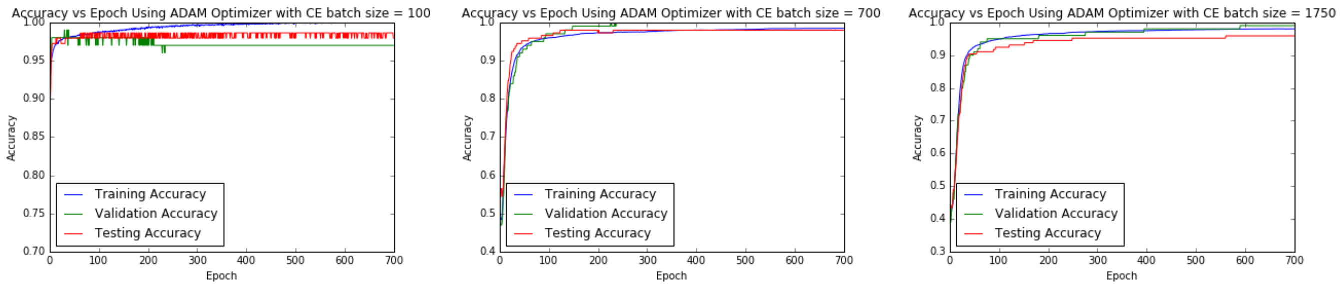


Figure 13: Training, validation and test accuracy using the ADAM optimizer and stochastic gradient descent at α=0.001, λ=0 and batch sizes of 100, 700, 1750 vs number of epochs.

Comments:

- Like before we retain very similar results when changing the batch sizes and using cross entropy.
- One thing to note is the losses converge to zero much faster in cross entropy and eventhough as batch sizes increase the accuracies still reach very close to 100 percent unlike MSE. This proves that for our dataset, cross entropy is a better method to use over MSE.
- Another important factor to note is when using cross entropy our accuracy increased but was stable from epoch to epoch unlike MSE.

3.5.4.A Investigation of impact of hyperparamaters on ADAM optimizer - β_1 - Cross Entropy Loss

Below are the plots of training, validation, and test accuracy and the effect of modulating hyperparameter β_1 :

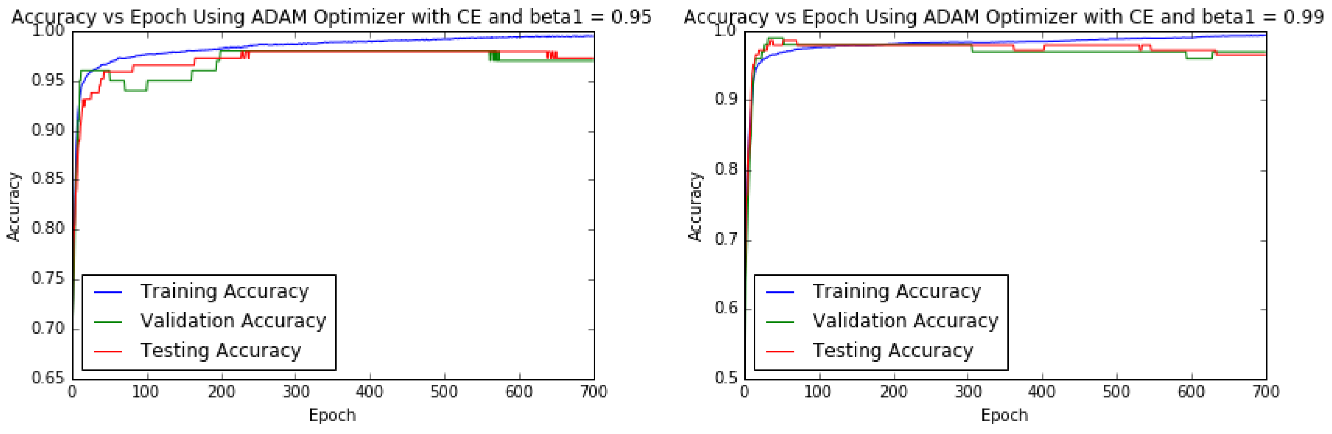


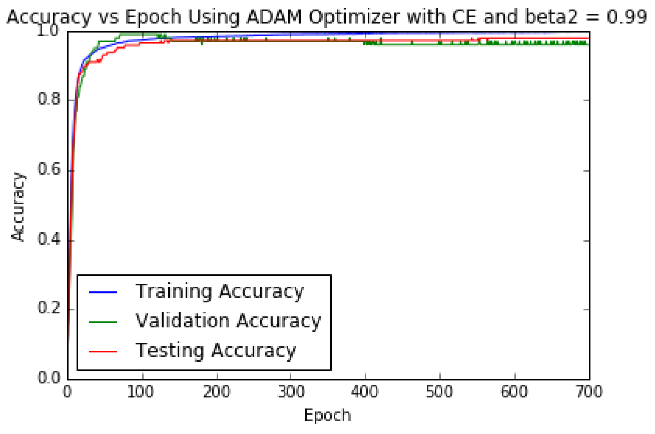
Figure 14: Cross-entropy training, validation and test loss using the ADAM optimizer and stochastic gradient descent at α=0.001, λ=0 and modulating β_1 value vs number of epochs.

Comments:

- We see how varying β_1 for cross entropy does not cause a significant variation in accuracies whereas in MSE it did.

3.5.4.B Investigation of impact of hyperparamaters on ADAM optimizer - β_2 - Cross Entropy Loss

Below are the plots of training, validation, and test accuracy and the effect of modulating hyperparameter β_2 :



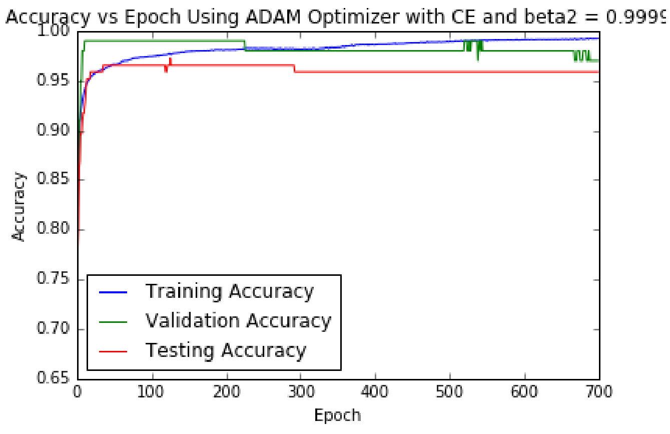


Figure 15: Cross-entropy training, validation and test loss using the ADAM optimizer and stochastic gradient descent at α=0.001, λ=0 and modulating β_2 value vs number of epochs.

Comments:

- We see again how varying β_2 has very little effect on the accuracies when using cross entropy.

3.5.4.C Investigation of impact of hyperparamaters on ADAM optimizer - ϵ - Cross Entropy Loss

Below are the plots of training, validation, and test accuracy and the effect of modulating hyperparameter ϵ :

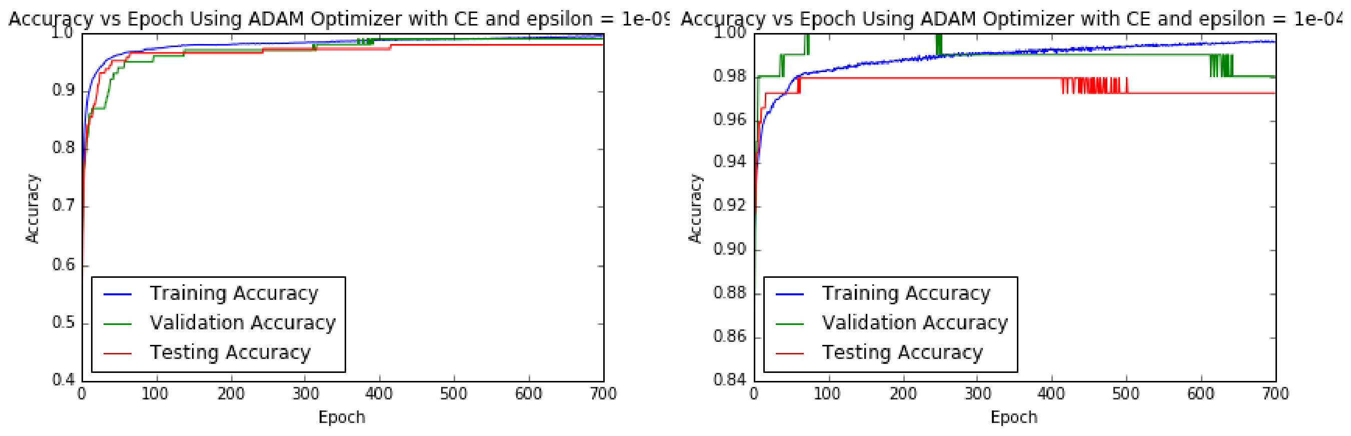


Figure 16: Cross-entropy training, validation and test loss using the ADAM optimizer and stochastic gradient descent at α=0.001, λ=0 and modulating ϵ value vs number of epochs.

Comments:

- The difference in epsilon produces very little difference in the final results for accuracies and looking at the graph it looks like when epsilon was 1e-04 the accuracies were very unstable but as you can see the scales for the two graphs are different, the graph for epsilon 1e-04 is much more zoomed in.

3.6 Comparing ADAM Optimizer to Batch Gradient Descent

From the above results it is clear how the ADAM optimizer provides better results. It has a significantly better convergence time and accuracy. This was expected because ADAM allows more fine tuning for SGD which allows better performance. When we do SGD we fixed the learning rate, but ADAM tunes the learning parameter as it learns. One major advantage ADAM has is the use of beta one and beta two, this allows it to tune decay of each moment accordingly. This is important for image classification because most of our image is just one colour and not valuable to us, therefore we can increase the rate in one direction much faster than the other hence we are able to learn faster. In conclusion, the ADAM optimizer is better because is it fine tunes its arguments as it learns, and because of its flexibility it is able to learn more efficiently.

Gradient derivation

Thursday, October 10, 2019 11:24 PM

$$MSE = \frac{1}{N} \sum_{n=1}^{N} || XW + b - Y ||_2^2 + \frac{\lambda}{2} ||W||_2^2$$

$$\nabla_W MSE = \partial/\partial W \, MSE = \frac{2}{N} \sum_{n=1}^{N} (XW + b - Y) X + \lambda W$$

$$= 2/N \Big(\sum_{n=1}^{W} XW + b - Y \Big) X + \lambda W$$

$\hookrightarrow$ Scalar value

$$\nabla_b MSE = \partial/\partial b \, MSE = 2/N \sum_{n=1}^{N} || XW + b - Y || \rightarrow \text{Scalar value.}$$

Gradient Derivation CE

Thursday, October 10, 2019 11:33 PM

$$CE = \frac{1}{N}\sum_{n=1}^{N}\left[-y_n \log(\hat{y}_n(x_n)) - (1-y_n)\log(1-\hat{y}(x_n))\right] + \frac{\lambda}{2}\|W\|_2^2$$

$$\hat{y}(x) = \sigma[W^Tx+b] \quad \text{or} \quad \sigma[xW+b]$$

$$\sigma(z) = \frac{1}{1+e^{-z}} = (1+e^{-z})^{-1}$$

$$\sigma'(z) = -1(1+e^{-z})^{-2}\cdot(-1)\cdot e^{-z}$$

$$= \frac{e^{-z}}{(1+e^{-z})^2} \qquad \text{Recall: } e^{-z} = (1+e^{-z})-1$$

$$= \frac{(1+e^{-z})}{(1+e^{-z})^2} - \frac{1}{(1+e^{-z})^2}$$

$$= \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2}$$

$$= \sigma(z) - \sigma^2(z) = \sigma(z)(1-\sigma(z))$$

$$\therefore \hat{y}(x) = \sigma(xW+b)$$

$$\hat{y}'(x) = x\cdot\sigma'(xW+b)$$

$$= \sigma(xW+b)(1-\sigma(xW+b))\cdot x$$

$$\therefore \nabla_W CE = \partial/\partial W\, CE = \frac{1}{N}\left[\sum_{n=1}^{N}(-y_n)\cdot\frac{\hat{y}'(x_n)}{\hat{y}(x_n)} - (1-y_n)\frac{-\hat{y}'(x_n)}{1-\hat{y}(x_n)}\right] + \lambda W$$

$$= \frac{1}{N}\left[\sum_{n=1}^{N}(-y_n)\cdot(1-\sigma(xW+b))\cdot x - (1-y_n)(-1)\cdot\sigma(xW+b)\cdot x\right] + \lambda W$$

$$= \frac{1}{N}\left[\sum_{n=1}^{N} xy_n\cdot\sigma(xW+b) - xy_n + x\cdot\sigma(xW+b) - xy_n\sigma(xW+b)\right] + \lambda W$$

$$= \frac{1}{N}\left[\sum_{n=1}^{N}(\sigma(xW+b)-y_n)x_n\right] + \lambda W$$

$$\nabla_b CE = \frac{1}{N}\sum_{n=1}^{N}(\sigma(xW+b)-y_n) \qquad \text{Same derivation as above.}$$