1. Linear Regression [18 points]

1. Loss Function and Gradient [4 pts]:

- Implement two vectorized Numpy functions to compute the loss and gradient respectively.Both functions should accept 5 arguments - the weight matrix, the bias matrix, the data matrix, the labels, and λ, the regularization parameter. The loss function return a number (indicating total loss). The gradient function should be an analytical expression of the loss function and return the gradient with respect to the weights and the gradient with respect to the biases. Both function headers are below. Include both the analytical expression for the gradient and the Python code snippet in your report.

Description:

The loss function of MSE as following:

$$L = \sum_{n=1}^{N} \frac{1}{2N} \| W^T x^{(n)} + b - y^{(n)} \|_2^2 + \frac{\lambda}{2} \|W\|_2^2$$

The gradient with respect to the weight as following:

$$\frac{\partial L}{\partial W_i} = \sum_{n=1}^{N} \frac{1}{\cancel{2}N} \sum_{i=1}^{m=784} \left(W_i x_i^{(n)} + b - y^{(n)} \right) \cdot \cancel{2} + \sum_{i=1}^{m=784} \frac{\lambda}{\cancel{2}} \cdot \cancel{2}\, W_i$$

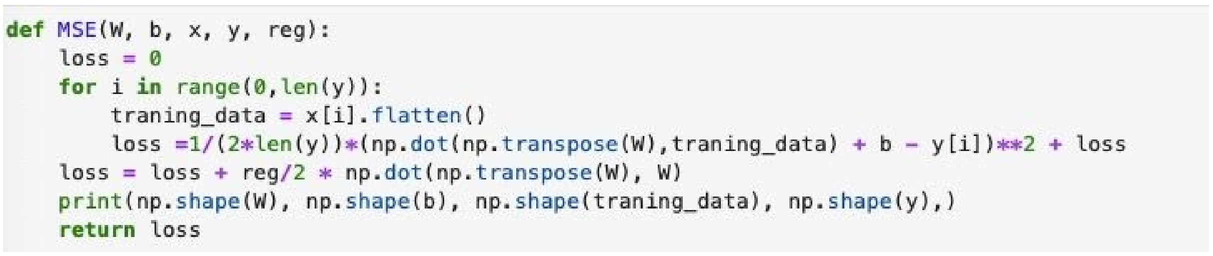
$$= \sum_{n=1}^{N} \frac{1}{N} \sum_{i=1}^{m=784} \left(W_i x_i^{(n)} + b - y^{(n)} \right) + \sum_{i=1}^{m=784} \lambda W_i$$

$$\frac{\partial L}{\partial W} = \sum_{n=1}^{N} \left(W x^{(n)} + b - y^{(n)} \right) + \lambda W$$

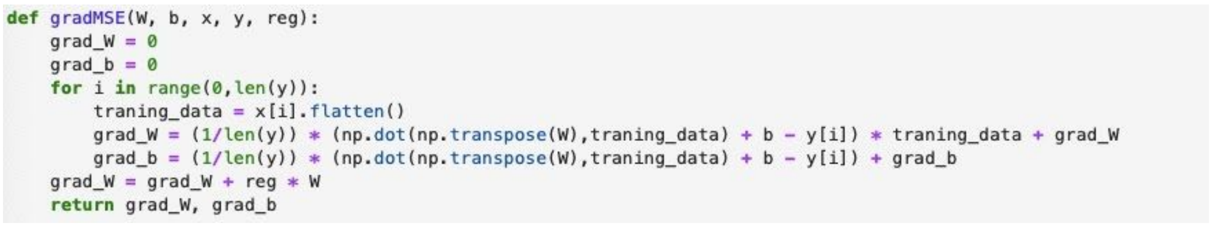
The gradient with respect to the bias as following:

$$\frac{\partial L}{\partial b} = \sum_{n=1}^{N} \frac{1}{N} \left(W^T x^{(n)} + b - y^{(n)} \right)$$

Python code snippet:



```python
def MSE(W, b, x, y, reg):
    loss = 0
    for i in range(0,len(y)):
        traning_data = x[i].flatten()
        loss =1/(2*len(y))*(np.dot(np.transpose(W),traning_data) + b - y[i])**2 + loss
    loss = loss + reg/2 * np.dot(np.transpose(W), W)
    print(np.shape(W), np.shape(b), np.shape(traning_data), np.shape(y),)
    return loss
```

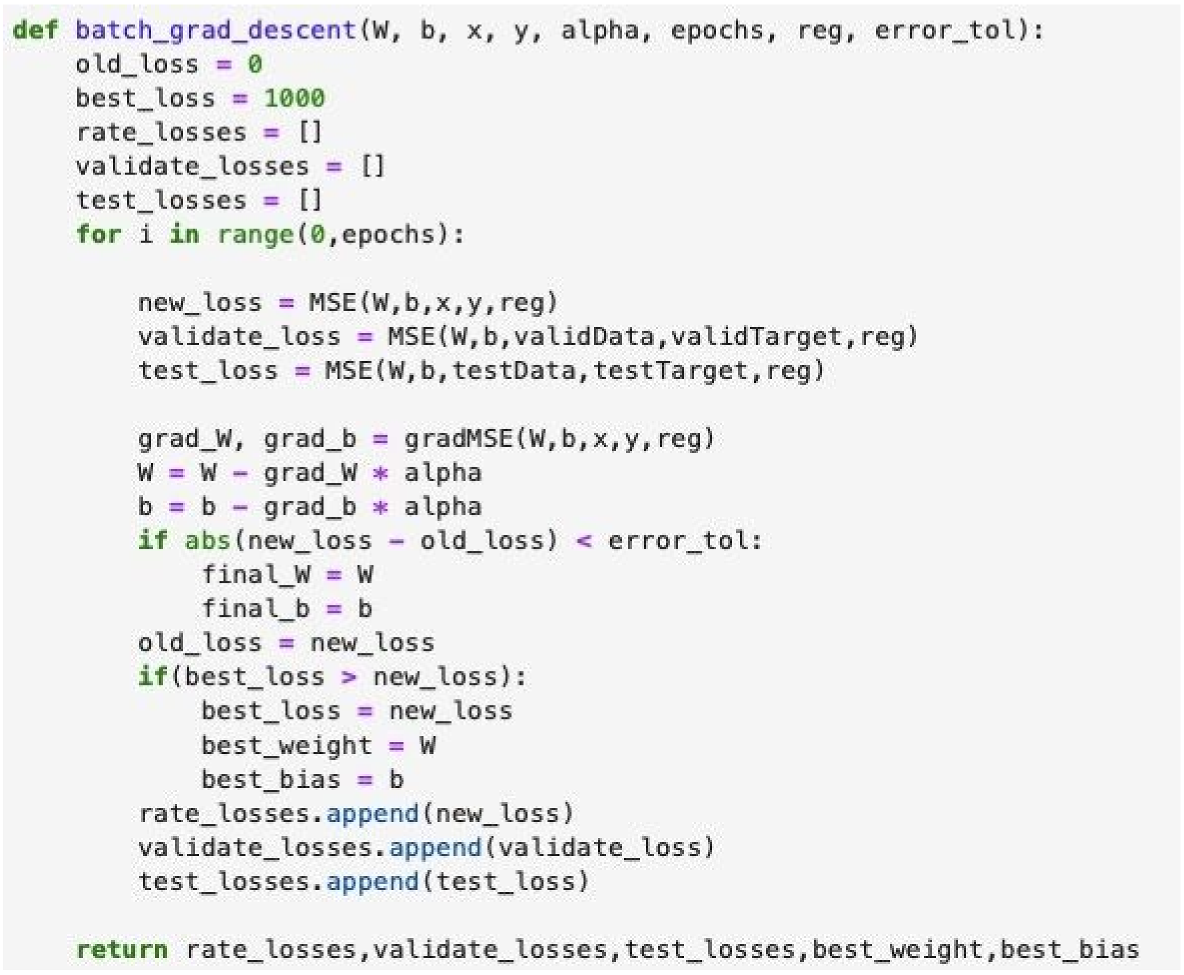


```python
def gradMSE(W, b, x, y, reg):
    grad_W = 0
    grad_b = 0
    for i in range(0,len(y)):
        traning_data = x[i].flatten()
        grad_W = (1/len(y)) * (np.dot(np.transpose(W),traning_data) + b - y[i]) * traning_data + grad_W
        grad_b = (1/len(y)) * (np.dot(np.transpose(W),traning_data) + b - y[i]) + grad_b
    grad_W = grad_W + reg * W
    return grad_W, grad_b
```

2. Gradient Descent Implementation [6 pts]:

- Using the gradient computed from Part A, implement the batch Gradient Descent algorithm to classify the two classes in the notMNIST dataset. The function should accept 8 arguments- the weight matrix, the bias matrix, the data matrix, the labels, the learning rate, the number of epochs 1 , λ and an error tolerance (set to 1×10^{-7}). The error tolerance will be used to compute the difference between the old and updated weights per iteration. The function should return the optimized weight and bias matrices. The function header is below.

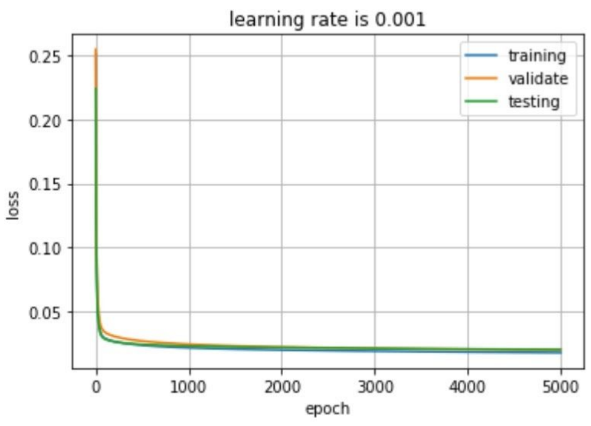
Python code snippet:

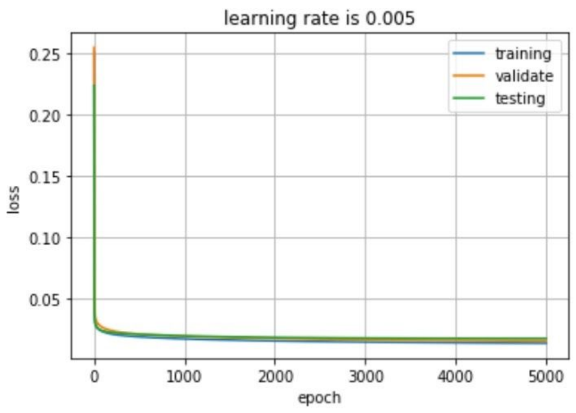


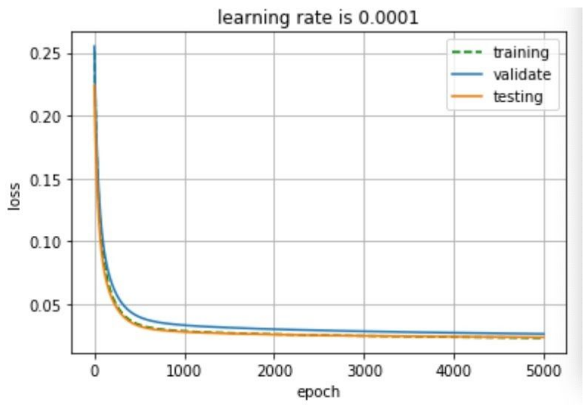
```python
def batch_grad_descent(W, b, x, y, alpha, epochs, reg, error_tol):
    old_loss = 0
    best_loss = 1000
    rate_losses = []
    validate_losses = []
    test_losses = []
    for i in range(0,epochs):

        new_loss = MSE(W,b,x,y,reg)
        validate_loss = MSE(W,b,validData,validTarget,reg)
        test_loss = MSE(W,b,testData,testTarget,reg)

        grad_W, grad_b = gradMSE(W,b,x,y,reg)
        W = W - grad_W * alpha
        b = b - grad_b * alpha
        if abs(new_loss - old_loss) < error_tol:
            final_W = W
            final_b = b
        old_loss = new_loss
        if(best_loss > new_loss):
            best_loss = new_loss
            best_weight = W
            best_bias = b
        rate_losses.append(new_loss)
        validate_losses.append(validate_loss)
        test_losses.append(test_loss)

    return rate_losses,validate_losses,test_losses,best_weight,best_bias
```

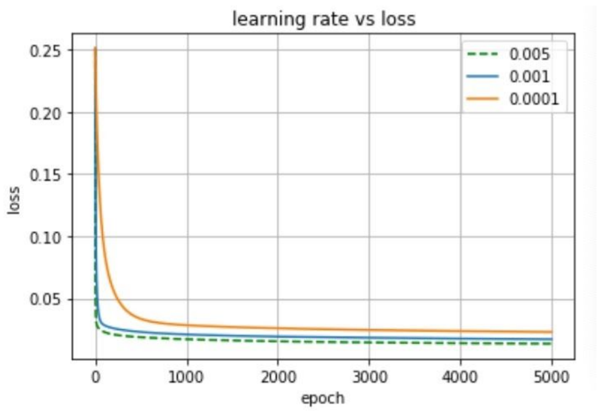
3. Tuning the Learning Rate[3 pts]:

- Test your implementation of Gradient Descent with 5000 epochs and λ = 0. Investigate the impact of learning rate, α = 0.005, 0.001, 0.0001 on the performance of your classifier. Plot the training, validation and test losses. What is the impact of modifying α against the training time? Final classification accuracy?









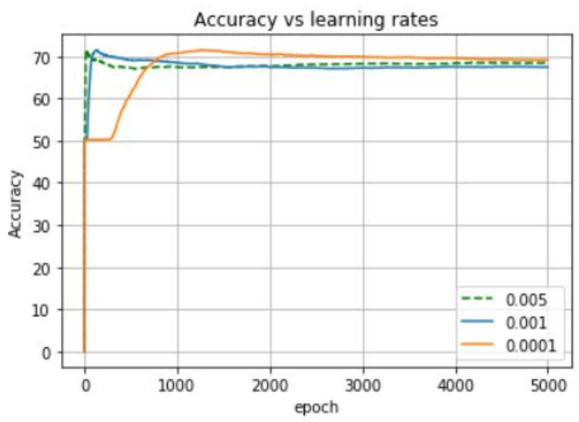


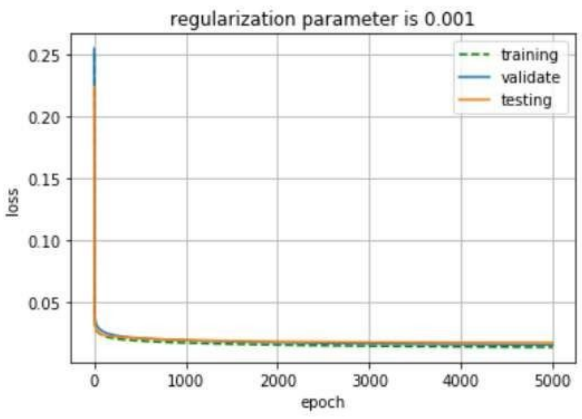
Table 1: Learning rate vs final MSE loss and Accuracy

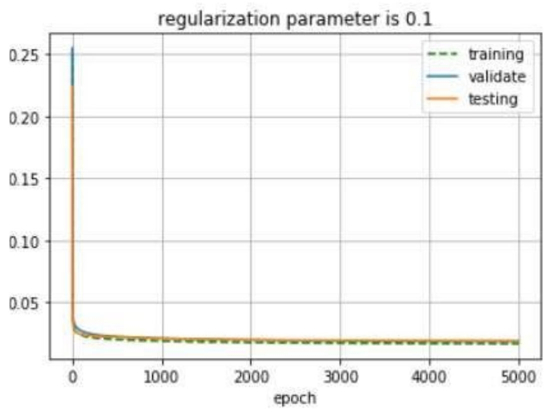
Learning rate	Final loss	Final Accuracy(%)
0.001	0.01721721	71.4
0.005	0.01365889	71.6
0.0001	0.02305699	71.4

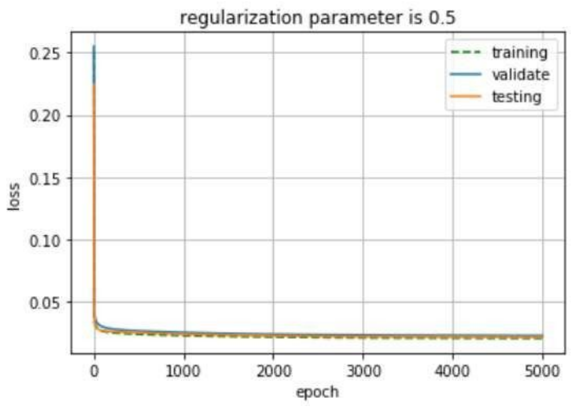
Description:
As shown in the graph of “learning rate vs loss“, the greater “alpha” has a faster speed of training. This trend is corresponding to the intuition of the learning rate. The curve of learning rate of 0.005 is steeper and faster than others but the curve of learning rate of 0.0001 is stable and may have a robust performance as this curve is smoother.
For the accuracy, the learning rate of 0.005 is better than others as well because it effectively decreases its errors and improves its weight and bias in a number of iterations

4. Generalization [3 pts]:

- Investigate impact by modifying the regularization parameter, $\lambda = \{0.001, 0.1, 0.5\}$. Plot the training, validation and test loss for $\alpha = 0.005$ and report the final training, validation and test performance of your classifier. Comment on the effect of regularization on performance as well as the rationale behind tuning λ using the validation set.







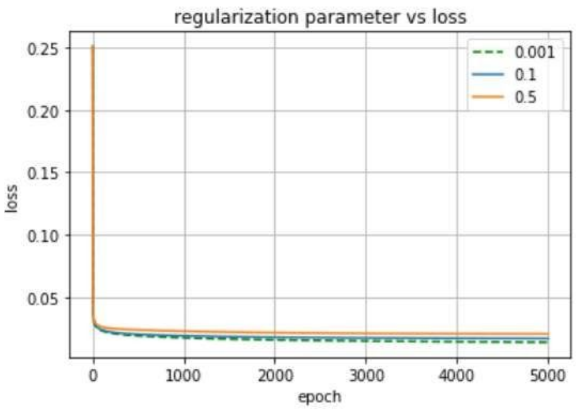


Table 2: Regression vs final MSE loss and Accuracy

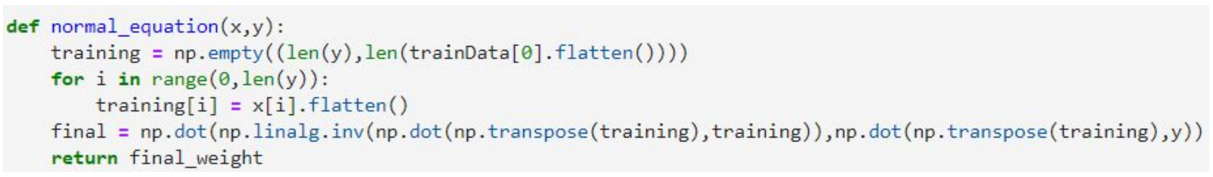
Regression parameter	Training	Validate	Testing
0.001	0.11452864	0.12664707	0.12742395
0.1	0.0556228	0.06149768	0.08577915
0.5	0.19781199	0.21539388	0.20092413

Description:
The best regression parameter is the smallest value, 0.001. The reason is that for the greater regression parameter, the loss function will be tolerated a greater penalty. The penalty is always used to solve the problem of overfitting that is the model should not be too close to the targets or labels.
The regression parameter is used to compare the performance of the different model, the validate dataset does not participate in the procedure of training so it is not fitting in the model. Generally, the validate dataset can be used to test the performance of the model by tuning the regression parameter, lambda.

5. Comparing Batch GD with normal equation [2 pts]:

- For linear regression, you can find the optimum weights using the closed form equation for the derivative of the means square error (normal equation). For zero weight decay, Write a Numpy script to find the optimal linear regression weights on the two-class notMNIST dataset using the "normal equation" of the least squares formula. Compare in terms of final training MSE, accuracy and computation time between Batch GD and normal equation.

Python code snippet:



```
def normal_equation(x,y):
    training = np.empty((len(y),len(trainData[0].flatten())))
    for i in range(0,len(y)):
        training[i] = x[i].flatten()
    final = np.dot(np.linalg.inv(np.dot(np.transpose(training),training)),np.dot(np.transpose(training),y))
    return final_weight
```

Table 3: Normal equation vs Batch gradient descent

Type	Time(s)	Accuracy(%)	Loss
Normal equation	0.0688	73.31	0.01158202
Batch GD	743.0811	71.6	0.01365889

Description:
The normal equation uses a much less time elapsed than the Batch GD, the accuracy and loss are fixed but it could be improved by adding one dimension into the matrix of weight which is the bias. The accuracy and loss are better than batch GD, but batch GD can be improved easily with more iterations. For the time complexity of the Batch GD, it can be approximated to O(m*n), m is the number of iterations and n is number of samples

2. Logistic Regression [18 points]

1. Loss Function and Gradient [4 pts]:

- Implement two vectorized Numpy functions to compute the Binary Cross Entropy Error and its gradient respectively. Similar to Part 1.1, both functions should accept 5 arguments - the weight matrix, the bias matrix, the data matrix, the labels, and the regularization parameter. They should return the loss and gradients with respect to weights and biases respectively. Both function headers are below. Include the analytical expressions in your report as well as a snippet of your Python code.

Description:
The loss function of MSE as following:

$$L = \sum_{n=1}^{N} \frac{1}{N}\left[y^{(n)} \ln\left(1+e^{-(w^T x^{(n)}+b)}\right) - (1-y^{(n)}) \ln\left(\frac{e^{-(w^T x^{(n)}+b)}}{1+e^{-(w^T x^{(n)}+b)}}\right)\right] + \frac{\lambda}{2}\|w\|_2^2$$

$$= \sum_{n=1}^{N} \frac{1}{N}\left[y^{(n)} \ln\left(1+e^{-(w^T x^{(n)}+b)}\right) - (1-y^{(n)})\left[\ln\left(e^{-(w^T x^{(n)}+b)}\right) - \ln\left(1+e^{-(w^T x^{(n)}+b)}\right)\right]\right] + \frac{\lambda}{2}\|w\|_2^2$$

$$= \sum_{n=1}^{N} \frac{1}{N}\left[y^{(n)} \ln\left(1+e^{-(w^T x^{(n)}+b)}\right) - (1-y^{(n)})(-w^T x^{(n)}-b) + (1-y^{(n)})\left(1+e^{-(w^T x^{(n)}+b)}\right)\right] + \frac{\lambda}{2}\|w\|_2^2$$

$$= \sum_{n=1}^{N} \frac{1}{N}\left[\ln\left(1+e^{-(w^T x^{(n)}+b)}\right) - (1-y^{(n)})(-w^T x^{(n)}-b)\right] + \frac{\lambda}{2}\|w\|_2^2$$

The gradient with respect to the weight as following:

$$\frac{\partial L}{\partial w_i} = \sum_{n=1}^{N} \frac{1}{N} \Big[\frac{1}{1+e^{-w^T x^{(n)} - b}} \frac{\partial}{\partial w_i} \big(1 + e^{-\sum_j w_j x_j^{(n)} - b}\big) - (1-y^{(n)}) \frac{\partial}{\partial w_i} \big(-\sum_j w_j x_j^{(n)}\big) \Big] + \lambda w_i$$

$$= \sum_{n=1}^{N} \frac{1}{N} \Big[\frac{1}{1+e^{-w^T x^{(n)} - b}} e^{-w^T x + b} \frac{\partial}{\partial w_i} \big(-\sum_j w_j x_j^{(n)} - b\big) - (1-y^{(n)}) \frac{\partial}{\partial w_i} \big(-\sum_j w_j x_j^{(n)}\big) \Big] + \lambda w_i$$

When $j = i$
$$= \sum_{n=1}^{N} \frac{1}{N} \Big[\frac{1}{1+e^{-w^T x^{(n)} - b}} e^{-w^T x + b} \cdot \frac{\partial}{\partial w_i} (-w_i x_i^{(n)}) - (1-y^{(n)}) \frac{\partial}{\partial w_i} (-w_i x_i^{(n)}) + \lambda w_i$$

$$= \sum_{n=1}^{N} \frac{1}{N} \Big[\frac{1}{1+e^{-w^T x^{(n)} - b}} e^{-w^T x + b} \cdot (-x_i^{(n)}) - (1-y^{(n)}) (-x_i^{(n)}) \Big] + \lambda w_i$$

$$= \frac{1}{N} \sum_{n=1}^{N} \Big[\frac{-x_i^{(n)} e^{-w^T x^{(n)} - b}}{1+e^{-w^T x^{(n)} - b}} + (1-y^{(n)}) x_i^{(n)} \Big] + \lambda w_i$$

$$= \frac{1}{N} \sum_{n=1}^{N} \Big[(1-y^{(n)}) - \frac{e^{-w^T x^{(n)} - b}}{1+e^{-w^T x^{(n)} - b}} \Big] \cdot x_i^{(n)} + \lambda w_i$$

$$= \frac{1}{N} \sum_{n=1}^{N} \Big[(1-y^{(n)}) - \frac{e^{-w^T x^{(n)} - b}}{1+e^{-w^T x^{(n)} - b}} \Big] \cdot x^{(n)} + \lambda w$$

The gradient with respect to the bias as following:

$$\frac{\partial L}{\partial b} = \sum_{n=1}^{N} \frac{1}{N} \Big[\frac{1}{1+e^{-w^T x^{(n)} - b}} \frac{\partial}{\partial b} \big(1 + e^{-\sum_j w_j x_j^{(n)} - b}\big) - (1-y^{(n)}) (-w^T x^{(n)} - b) \Big]$$

$$= \sum_{n=1}^{N} \frac{1}{N} \Big[\frac{1}{1+e^{-w^T x^{(n)} - b}} e^{-w^T x} \frac{\partial}{\partial b} e^{-b} + (1-y^{(n)})$$

When $j = i$
$$= \sum_{n=1}^{N} \frac{1}{N} \Big[\frac{-1}{1+e^{-w^T x^{(n)} - b}} e^{-w^T x - b} + (1-y^{(n)}) \Big]$$

$$= \sum_{n=1}^{N} \frac{1}{N} \Big[\frac{1}{1+e^{-w^T x^{(n)} - b}} - y^{(n)} \Big) \Big]$$

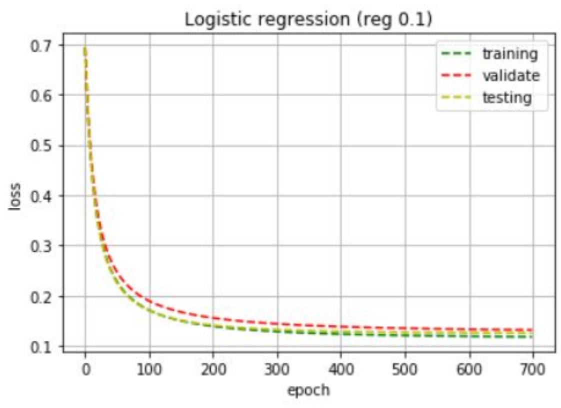
Python code snippet:

```
def crossEntropyLoss(W, b, x, y, reg):
    loss = 0
    for i in range(0,len(y)):
        traning_data = x[i].flatten()
        y_bar = np.dot(np.transpose(W),traning_data) + b
        loss =1/(len(y))*(np.log(1 + np.exp(-y_bar)) + (1-y[i])*y_bar) + loss
    loss = loss + reg/2 * np.dot(np.transpose(W), W)
    #print(y_bar)
    return loss
```

```
def gradCE(W, b, x, y, reg):
    grad_W = 0
    grad_b = 0.0
    for i in range(0,len(y)):
        traning_data = x[i].flatten()
        expo_fun = np.exp(-(np.dot(np.transpose(W),traning_data) + b))
        grad_W = (1/len(y)) * ((1-y[i]) - (expo_fun/(1+expo_fun))) * traning_data + grad_W
        grad_b = (1/len(y)) * (1/(1+expo_fun) - y[i]) + grad_b
    grad_W = grad_W + reg * W
    #print(grad_W)
    return grad_W, grad_b
```

2. Learning [4 pts]:

- Modify the function from Part 1.2 to include a flag, specifying the type of loss/gradient to use in the classifier. Modify your function to update weights and biases using the Binary Cross Entropy loss and report on the performance of the Logistic Regression model by setting λ = 0.1 and 5000 epochs. Plot the loss and accuracy curves.



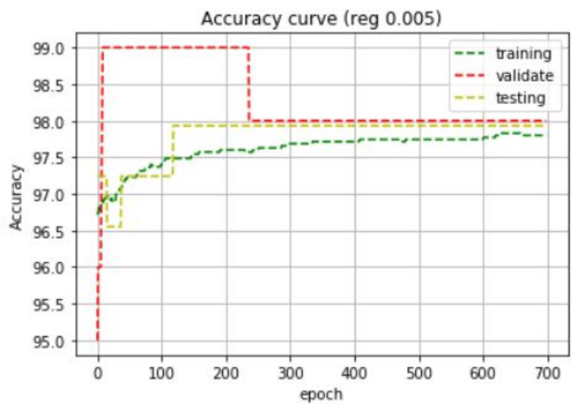
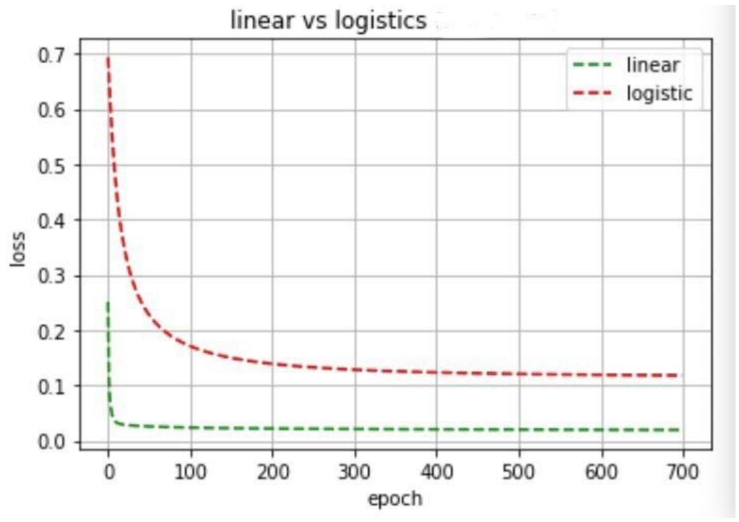


Table 4: CE loss and Accuracy

type	Final loss	Best Accuracy(%)
Training	0.01372	98.05
Validate	0.01672	99.0
Testing	0.02039	97.93

3. Comparison to Linear Regression [2 pts]:

- For zero weight decay, learning rate of 0.005 and 5000 epochs , plot the training cross entropy loss and MSE loss for logistic regression and linear regression respectively. Comment on the effect of cross-entropy loss convergence behaviour.



Description:
The convergence behavior for linear regression and logistics regression is different. For linear regression, it uses the mean square error to be the loss function, then its output of error will be enlarged, the gradient descent is very large in just a few epochs. However, in the logistics regression, it uses cross-entropy to be the loss function. In the cross-entropy, the gradient descent can keep fast descent other than mean square error which will be slower and slower. The slope of sigmoid is steep even the data is close to its upper bound and lower bound. In fact, usual loss functions will decrease their speed when they approach the bound, but cross-entropy is special because it is the logarithmic function, so it will remain that fast speed of descent.

3. Batch Gradient Descent vs. SGD and Adam [25 points]

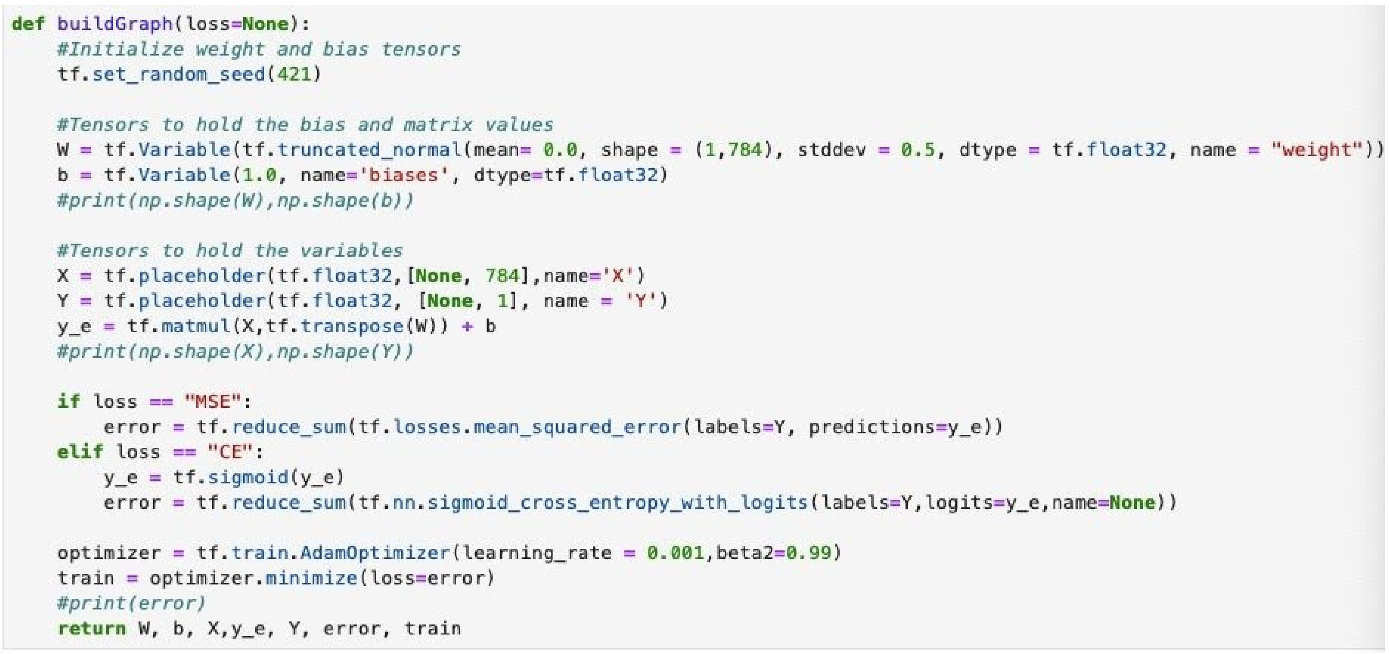
1. Building the Computational Graph [5 pts]:

- The function should return the weight, bias, predicted labels, real labels, the loss, the optimizer and the regularization parameter. The function header is below.

Description:

BuildGraph Function:

1. Initialize Tensors in Tensorflow. Set Weight and bias as variable tensors.
2. X and Y as input tensors. Meanwhile, calculate the estimation for y (W^TX+b).
3. Use mean_squared_error and sigmoid_cross_entropy_with_logits inside tensorflow to set for *loss* tensor.
4. Use AdamOptimizer machine learning to minimize the training error.

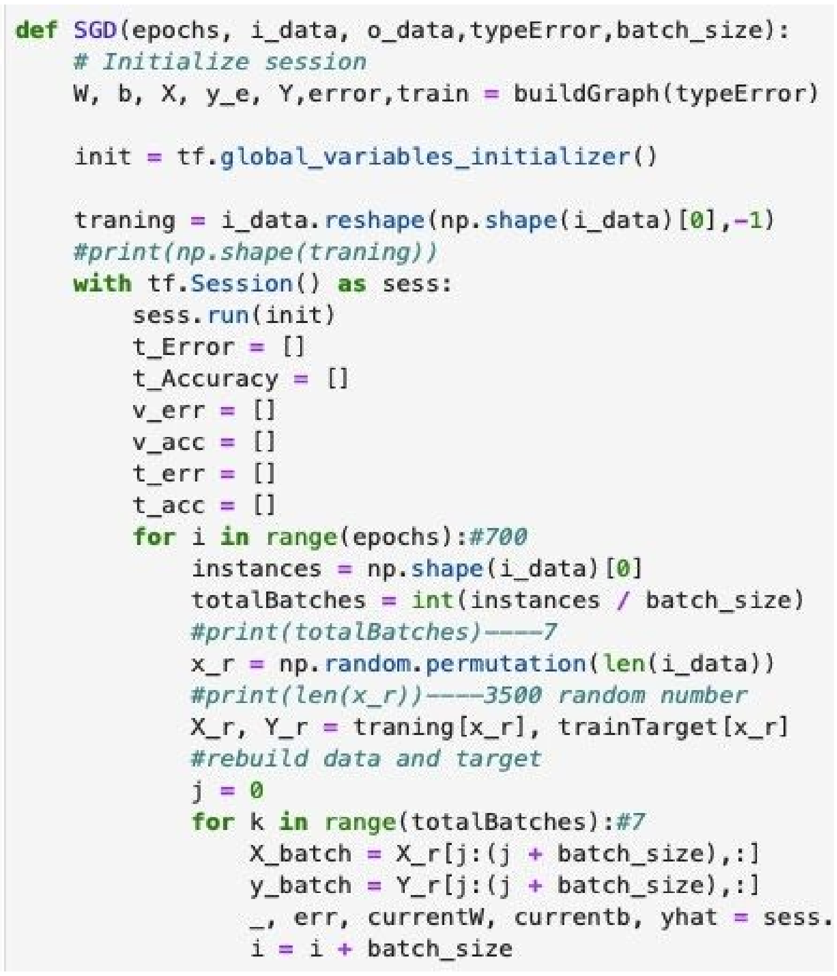


```python
def buildGraph(loss=None):
    #Initialize weight and bias tensors
    tf.set_random_seed(421)

    #Tensors to hold the bias and matrix values
    W = tf.Variable(tf.truncated_normal(mean= 0.0, shape = (1,784), stddev = 0.5, dtype = tf.float32, name = "weight"))
    b = tf.Variable(1.0, name='biases', dtype=tf.float32)
    #print(np.shape(W),np.shape(b))

    #Tensors to hold the variables
    X = tf.placeholder(tf.float32,[None, 784],name='X')
    Y = tf.placeholder(tf.float32, [None, 1], name = 'Y')
    y_e = tf.matmul(X,tf.transpose(W)) + b
    #print(np.shape(X),np.shape(Y))

    if loss == "MSE":
        error = tf.reduce_sum(tf.losses.mean_squared_error(labels=Y, predictions=y_e))
    elif loss == "CE":
        y_e = tf.sigmoid(y_e)
        error = tf.reduce_sum(tf.nn.sigmoid_cross_entropy_with_logits(labels=Y,logits=y_e,name=None))

    optimizer = tf.train.AdamOptimizer(learning_rate = 0.001,beta2=0.99)
    train = optimizer.minimize(loss=error)
    #print(error)
    return W, b, X,y_e, Y, error, train
```

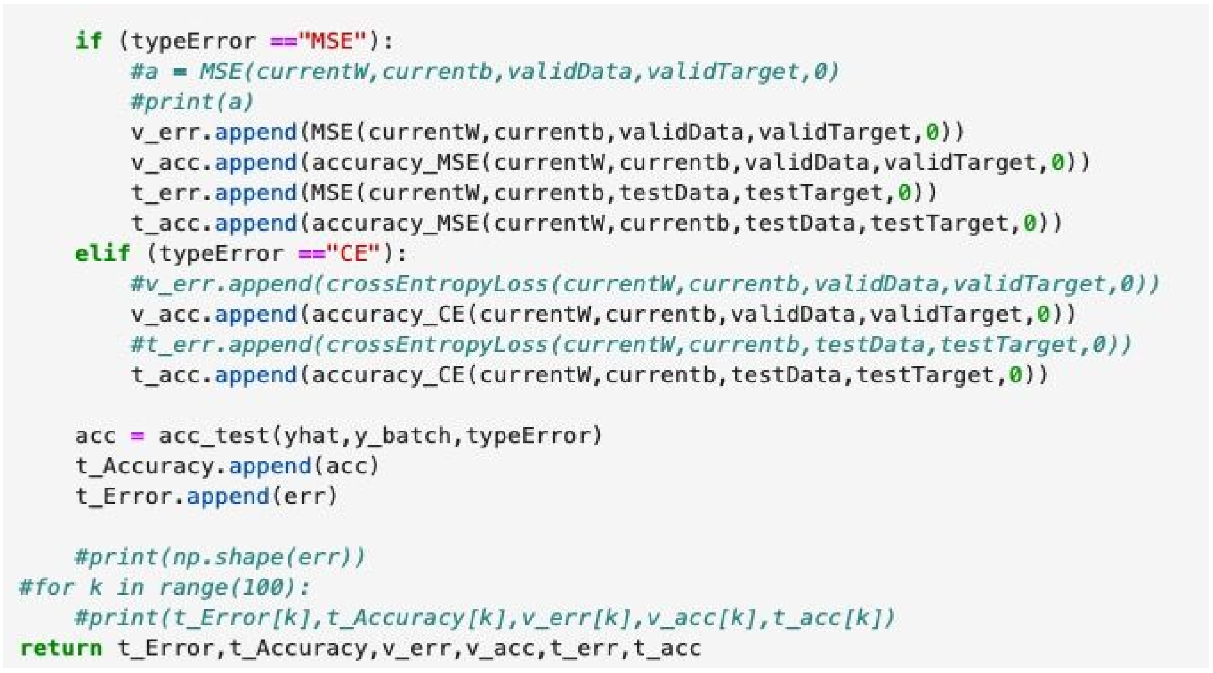
Code for buildGraph() function

Derivation for SGD algorithm:

- Use random function to shuffle the data set. Use minibatch size as 500 to batch the data set into 7 batches. After each epochs, reshuffle the training data and get the current Weight and bias. Then, calculate the accuracy for validation data and test data.
- After each epoch, store the training, validation and test losses and accuracies.



```python
def SGD(epochs, i_data, o_data,typeError,batch_size):
    # Initialize session
    W, b, X, y_e, Y,error,train = buildGraph(typeError)

    init = tf.global_variables_initializer()

    traning = i_data.reshape(np.shape(i_data)[0],-1)
    #print(np.shape(traning))
    with tf.Session() as sess:
        sess.run(init)
        t_Error = []
        t_Accuracy = []
        v_err = []
        v_acc = []
        t_err = []
        t_acc = []
        for i in range(epochs):#700
            instances = np.shape(i_data)[0]
            totalBatches = int(instances / batch_size)
            #print(totalBatches)----7
            x_r = np.random.permutation(len(i_data))
            #print(len(x_r))----3500 random number
            X_r, Y_r = traning[x_r], trainTarget[x_r]
            #rebuild data and target
            j = 0
            for k in range(totalBatches):#7
                X_batch = X_r[j:(j + batch_size),:]
                y_batch = Y_r[j:(j + batch_size),:]
                _, err, currentW, currentb, yhat = sess.
                i = i + batch_size
```



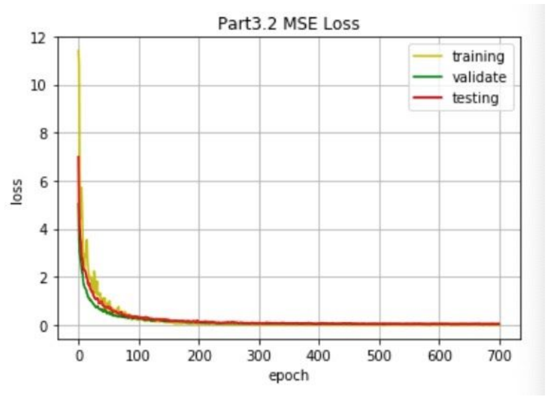
```python
    if (typeError =="MSE"):
        #a = MSE(currentW,currentb,validData,validTarget,0)
        #print(a)
        v_err.append(MSE(currentW,currentb,validData,validTarget,0))
        v_acc.append(accuracy_MSE(currentW,currentb,validData,validTarget,0))
        t_err.append(MSE(currentW,currentb,testData,testTarget,0))
        t_acc.append(accuracy_MSE(currentW,currentb,testData,testTarget,0))
    elif (typeError =="CE"):
        #v_err.append(crossEntropyLoss(currentW,currentb,validData,validTarget,0))
        v_acc.append(accuracy_CE(currentW,currentb,validData,validTarget,0))
        #t_err.append(crossEntropyLoss(currentW,currentb,testData,testTarget,0))
        t_acc.append(accuracy_CE(currentW,currentb,testData,testTarget,0))

    acc = acc_test(yhat,y_batch,typeError)
    t_Accuracy.append(acc)
    t_Error.append(err)

    #print(np.shape(err))
#for k in range(100):
    #print(t_Error[k],t_Accuracy[k],v_err[k],v_acc[k],t_acc[k])
return t_Error,t_Accuracy,v_err,v_acc,t_err,t_acc
```

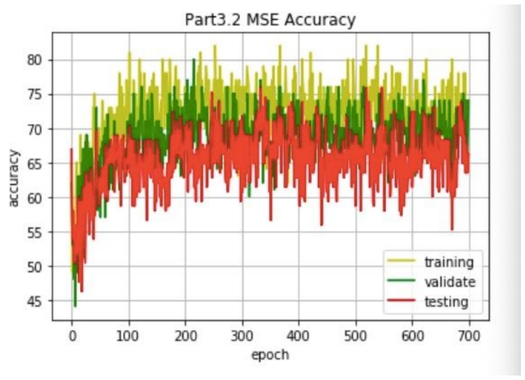
Code for SGD() function

2. Implementing Stochastic Gradient Descent [5 pts]:

- Implement the SGD algorithm for a minibatch size of 500 optimizing over 700 epochs 2 , minimizing the MSE. Calculate the total number of batches required by dividing the number of training instances by the minibatch size. After each epoch you will need to reshuffle the training data and start sampling from the beginning again. Initially, set λ = 0 and continue to use the same α value (i.e. 0.001). After each epoch, store the training, validation and test losses and accuracies. Use these to plot the loss and accuracy curves.

Description:



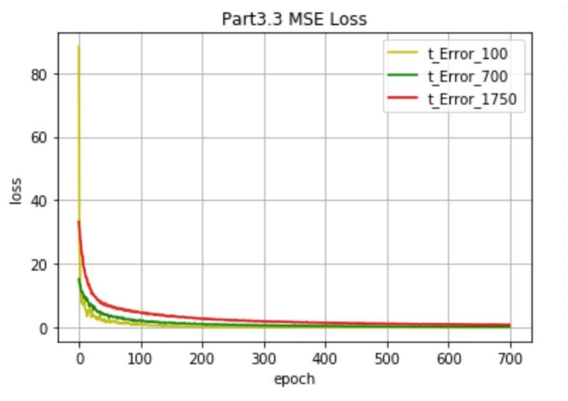


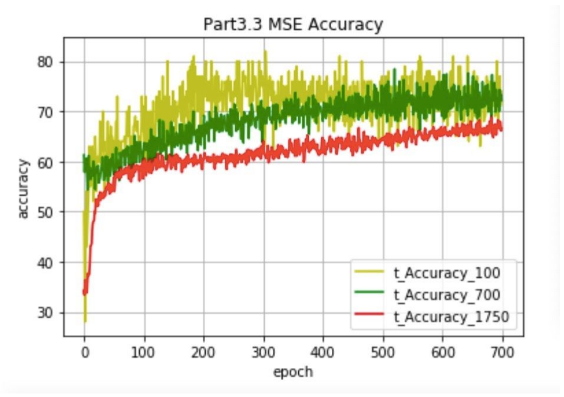
Observation: Number of batches: 7.
Training, validation and test losses and accuracies are stored.

3. Batch Size Investigation [2 pts]:

- Study the effects of batch size on behaviour of the SGD algorithm optimized using Adam by optimizing the model using batch sizes of B = {100, 700, 1750}. For each case, plot the loss and accuracy curves. What is the impact of batch size on the final classification accuracy for each of the 3 cases? What is the rationale for this?

Description:

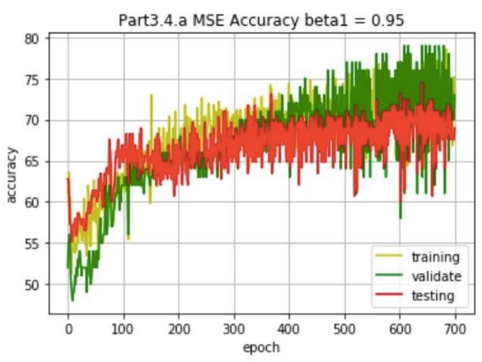




Observation: On these two graphs, when the batch size decreases, the loss becomes lower and the accuracy becomes larger. The machine learning becomes more powerful because the total training data have been sliced into small pieces. However, the running time for smaller batch size will become longer than larger batch size. Therefore, it is a trade-off between accuracy with running time.

4. Hyperparameter Investigation [4 pts.]:

- Experiment with the following Adam hyperparameters and for each, report on the final training, validation and test accuracies:
  (a) β1 = {0.95, 0.99}



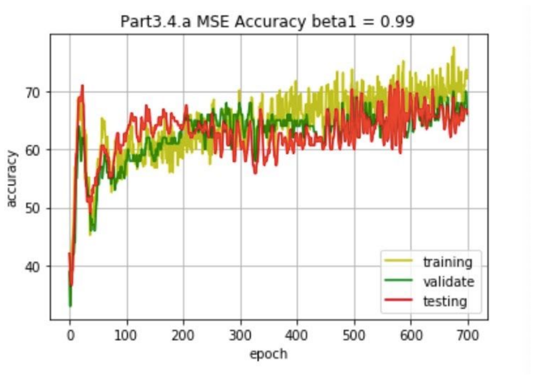
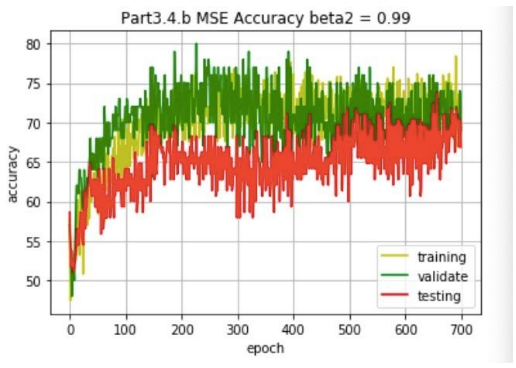


Figure. Accuracy graphs for beta1 = 0.95 and 0.99(default = 0.9)

Observation: As beta1 indicates the exponential decay rate for the 1st moment estimates, the output graph for beta1 = 0.99 shows that higher decay rate will result in faster learning with wider discrete distribution in small epochs. Also, the hyperparameter beta1 control the moving average of exponential decay rate.

$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \text{ (Update biased first moment estimate)}$$

(b) β2 = {0.99, 0.9999}



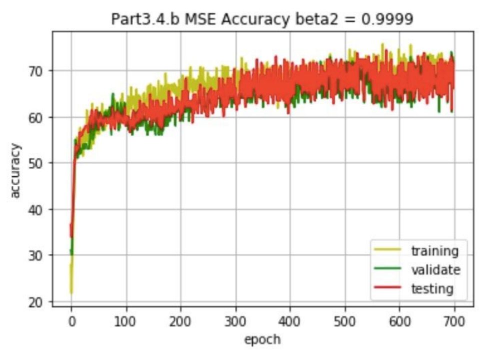


Figure. Accuracy graphs for beta2 = 0.99 and 0.9999(default = 0.999)

Observation: As beta2 indicates the exponential decay rate for the 2nd moment estimates, the output graph for beta2 = 0.9999 has tighter distribution for accuracy. This indicates that higher decay rate for second-order matrix results in better bias-correction due to the hyperparameter beta2 control the sparse gradient.

$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \text{ (Update biased second raw moment estimate)}$$

$$v_t = (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot g_i^2$$

(c) $= \{1e-09, 1e-4\}$

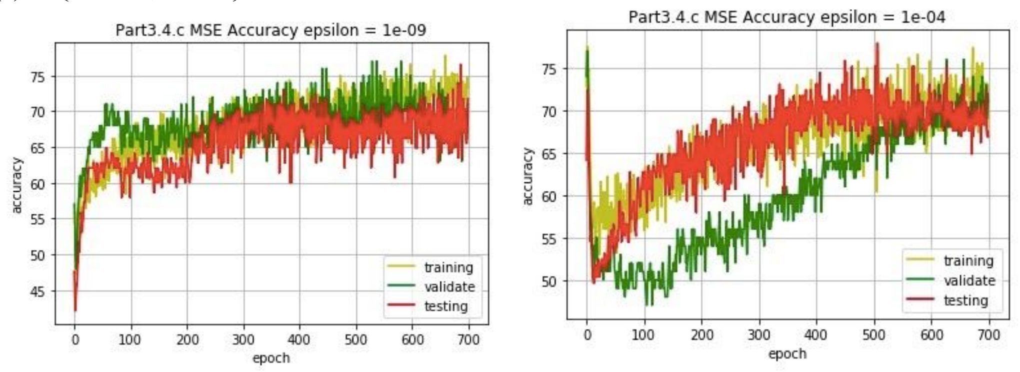
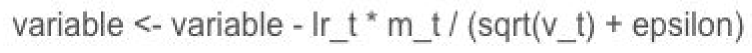


Figure. Accuracy graphs for epsilon = 1e-09 and 0.1e-04(default = 1e-08)

Observation: As epsilon represents a small constant for numerical stability, this constant help Adamoptimizer to reduce the case divided by 0. The output accuracy will become unstable as the constant added become larger (1e-04 is much larger than 1e-09). However, the accuracy will return to stable as epochs become more.



```
variable <- variable - lr_t * m_t / (sqrt(v_t) + epsilon)
```

For this part, use a minibatch size B = 500 and a learning rate of α = 0.001 and optimize over 700 epochs. For each of the three hyperparameters listed above, keep the other two as the default Tensorflow initialization. For each, what is the hyperparameter impact on the final training, validation and test accuracy? Why is this happening?

Description: Give different hyperparameters in Adam Optimizer

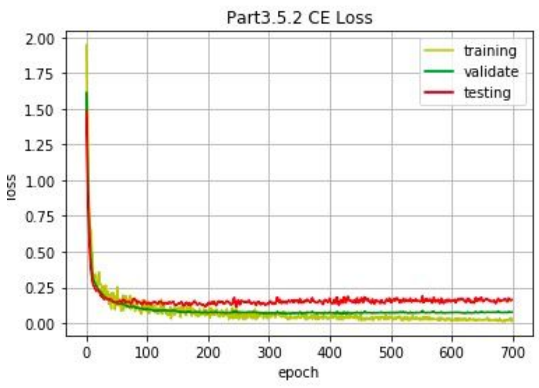
5. Cross-Entropy Loss Investigation [6 pts.]:

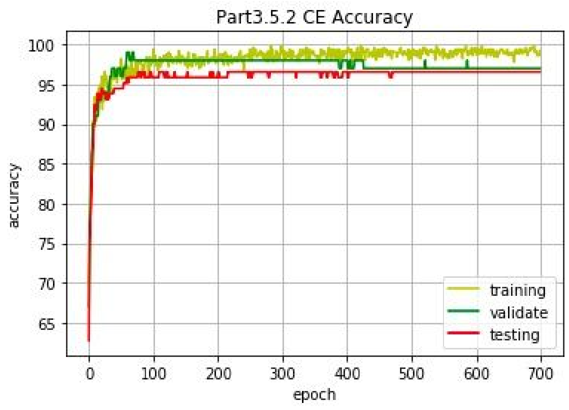
- Repeat parts 3.1.2 to 3.1.4 by minimizing the binary cross entropy loss. How do the two models compare against each other in terms model performance (i.e. final classification accuracy)?

Description:

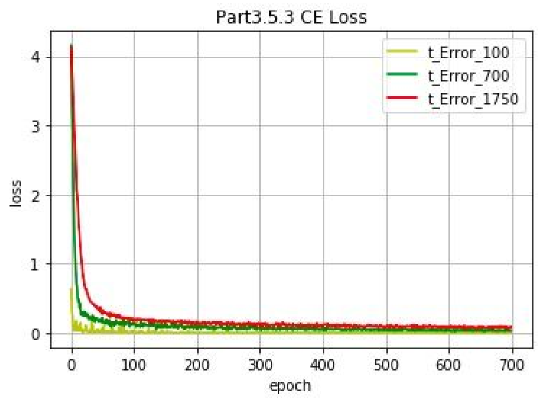
Performance for 3.1.2 to 3.1.4 for Cross-Entropy Loss

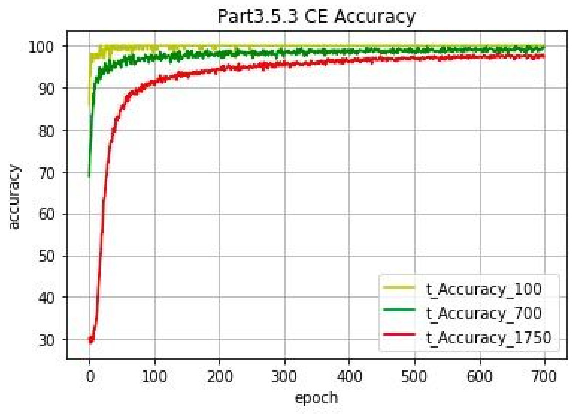
CE: 3.2



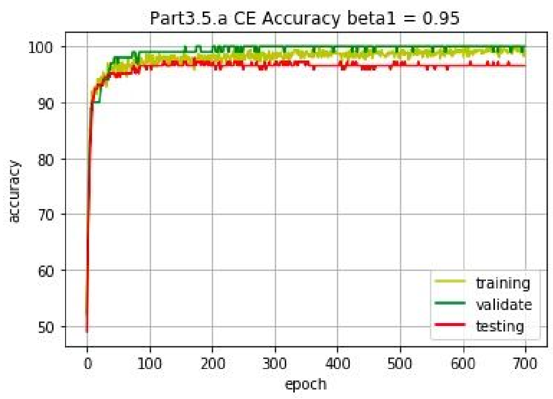


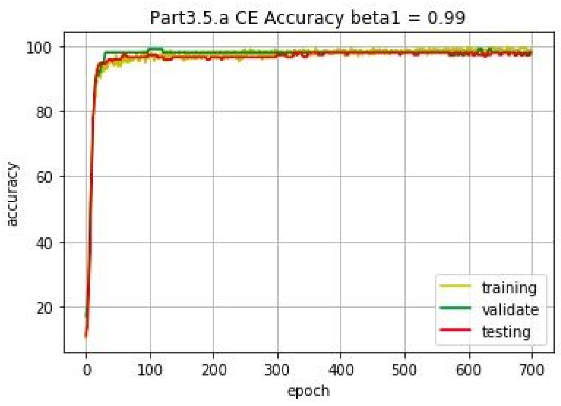
3-3



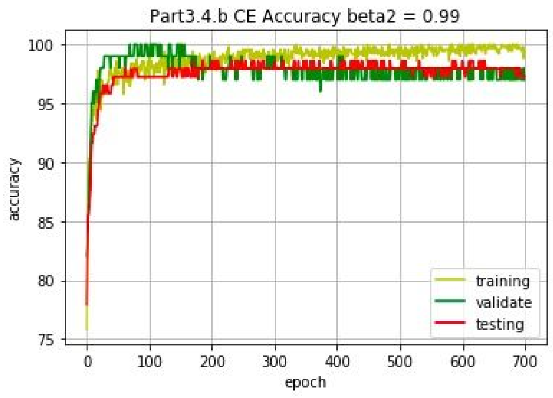


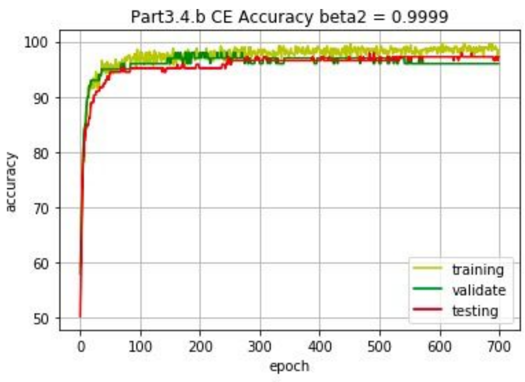
3-4-a



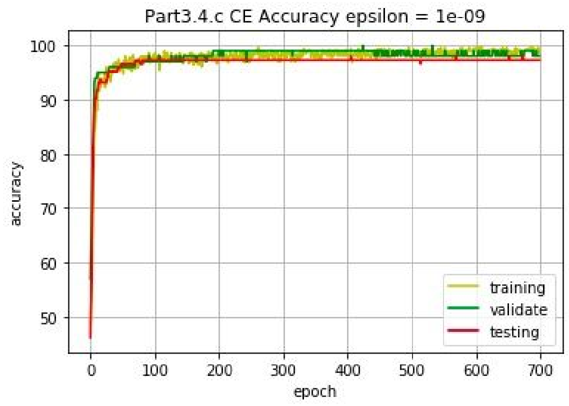


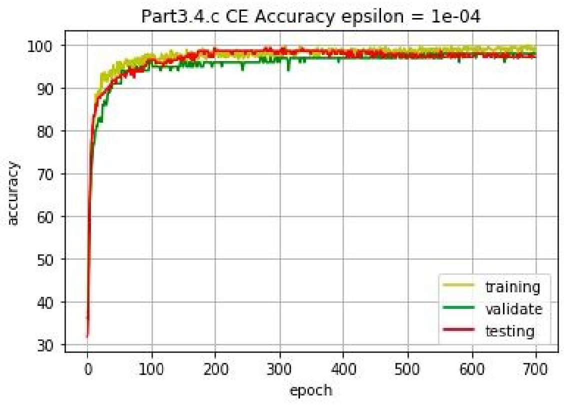
3-4-b





3-4-c





Observation: For this part, the conclusions for MSE mentioned above(conclusions for batch size, hyperparameters - beta1, beta2, epsilon) still hold.
Batch size: control the learning speed, smaller batch size results in faster approach to final accuracy while losing speed in running time.
Beta1: control the moving average of exponential decay rate.
Beta2: control bias-correction and the sparse gradient.
Epsilon: constant for numerical stability.
Totally, compared outputs from Cross Entropy(CE) loss with outputs from Mean Square Error(MSE) loss, it indicates that the accuracy of CE(avg98%) is larger than the accuracy of MSE(avg76%).

6. Comparison against Batch GD [3 pts.]:

- Comment on the overall performance of the SGD algorithm with Adam vs. the batch gradient descent algorithm you implemented earlier. Additionally, comment on the plots of the losses and accuracies of the SGD vs. batch gradient descent implementation. What do you notice about the curves? Why is this happening?

Discussion: After implementing both SGD algorithm with Adam and batch gradient descent algorithm, they both use batch gradient descent to learning the data. However, the SGD algorithm shuffle the data at the beginning of each epoch while the BGD use the unique data set from inputs. From the curves from loss and accuracy, the curves of BGD is more smooth than the curves of SGD because SGD algorithm will generate more possibilities after each epoch. In another word, the SGD algorithm will cover more possibilities than BGD algorithm. As the number of epochs increases, the results(loss and accuracy) from SGD will be more powerful.