Can Kocagil
21602218

Bilkent University

Department of Electric and Electrical Engineering

EEE443/543 Neural Networks

Mini Project 2

13.11.2020

TABLE OF CONTENTS

1 QUESTION 1

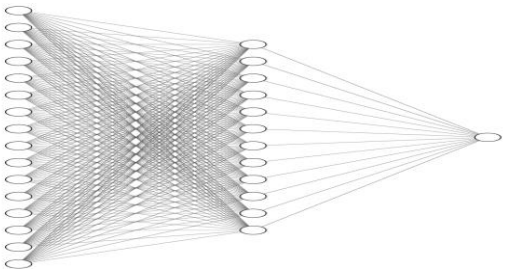
In question 1, there is a binary classification task on visual images with cat versus cars. Stochastic gradient descent should be performed on the mini batches. The evaluation metrics for this problem are mean squared error and mean classification error that should be recorded based on the epoch number.

1.1 PART A

In part a, our task is to design a multi-layer neural network with a single hidden layer using the backpropagation algorithm assuming a hyperbolic tangent activation function for all neurons. Hyperparameter tuning should be implemented to find optimal values for configurable parameters.

1.1.1 Architecture and Dataset

In this question, I am going to design multi-layer perceptron with a single hidden layer. The following neural architectures shows the overall structure.



Input layer has a dimension $\mathbb{R}^{1024}$, even the hidden layer neuron unit is a hyperparameter, I tune the neurons so that result will be $\in \mathbb{R}^{19}$ that we will see in the following parts and lastly output layer $\in \mathbb{R}^{1}$ since the task is binary classification cats versus cars. The input has 1900 sample and each sample has a dimension $\mathbb{R}^{1024}$.

The dataset consist of 1900 traning samples with 32 x 32 pixels in gray scale so that color channel is 1. To assess the performance of the model, there is a testing set with 1000 samples. Moreover dataset has balanced classes.(i.e., 950 cat and 950 car).

1.1.2 Image Preprocessing

To feed the inputs to the network, I flatten and normalize the data. The following code describes the flattening and normalizing the image input.

```
def flatten_images(self,X):
    x = X.reshape(X.shape[0],-1)
    return x

def normalize(self,X):
    return X/255
```

1.1.3 Configurable Parameters

In this question, our hyperparameters are number of hidden layer neurons, batch size, learning rate and initialization of the weights W_1, W_2 and biases B_1, B_2 assuming a hyperbolic tangent activation function for all neurons. To do that, I dediced to change the parameters according to followin intervals:

- Batch size ϵ {2,4,8,16,32,64}
- Hidden layer unit h ϵ {19,38,76,108}
- Learning rate η ϵ {0.1,..., 0.5}
- Weight W_1, W_2 and B_1, B_2 are initialized where W_1, B_1 and W_2, B_2 corresponds to the hidden layers and output layer, respectively using following initialization techniques:

1. ***Gauss Initialization*** $\sim \mathcal{N}(\mu, \sigma^2)$ where μ is mean and σ^2 is variance.
2. ***Xavier Initialization***

   Xavier Initialization, or Glorot Initialization, is an initialization scheme for neural networks. Biases are initialized be 0 and the weights W_{ij} at each layer are initialized as:

$$W_{ij} \sim Uniform[-\frac{1}{\sqrt{n_i}}, \frac{1}{\sqrt{n_j}}]$$

   or can be viewed as

$$W_{ij} \sim \mathcal{N}(\mu = 0, \sigma^2 = 1) * \frac{1}{\sqrt{n_i + n_j}}$$

   where n_i and n_o is the size of the previous layer and following layer respectively. Note that in glorot initialization there may be other formats.

3. ***He Initialization***

   He Initialization, is an initialization scheme for neural networks. Biases and the weights W_{ij} at each layer are initialized as:

$$W_{ij} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{2}{n})$$

where n is the size of the previous layer.

The implementation of above is on the following Python code:

```
class TwoLayerNetwork:

    def __init__(self,input_size = X_train.shape,batch_size = 19
,n_neurons = 76 , mean = 0,std = 1, lr = 1e-1, distribution = 'Xavier'):
        np.random.seed(15)
        self.lr = lr
        self.mse_train = {}
        self.mce_train = {}
        self.mse_test = {}
        self.mce_test = {}

        self.sample_size = input_size[0]
        self.feature_size = input_size[1]
        self.batch_size = batch_size
        self.n_neurons = n_neurons
        self.mean,self.std = mean,std

        self.dist = distribution


        self.n_update = round((self.sample_size/self.batch_size))

        self.W1_size = self.feature_size,self.n_neurons
        self.W2 size = self.n neurons,1

        self.B1_size = 1,self.n_neurons
        self.B2_size = 1,1

        self.B1 = Gauss(loc = self.mean, scale = self.std, size =
(self.B1_size)) * 0.01
        self.B2 = Gauss(loc = self.mean, scale = self.std, size =
(self.B2_size)) * 0.01


        self.he_scale1 = np.sqrt(2/self.feature_size)
        self.he_scale2 = np.sqrt(2/self.n_neurons)
        self.xavier_scale1 =
np.sqrt(2/(self.feature_size+self.n_neurons))
        self.xavier_scale2 = np.sqrt(2/(self.n_neurons+1))



        if (self.dist == 'Zero') :
            self.W1 = np.zeros((self.W1_size))
            self.W2 = np.zeros((self.W2_size))

        elif (self.dist == 'Gauss'):
            self.W1 = Gauss(loc = self.mean, scale = self.std, size =
(self.W1_size))* 0.01
            self.W2 = Gauss(loc = self.mean, scale = self.std, size =
(self.W2_size))* 0.01

        elif (self.dist == 'He'):
```

```
            self.W1 = randn(self.W1_size[0],self.W1_size[1]) *
self.he_scale1
            self.W2 = randn(self.W2 size[0],self.W2 size[1]) *
self.he_scale2

        elif (self.dist == 'Xavier'):

            self.W1 = randn(self.W1_size[0],self.W1_size[1]) *
self.xavier_scale1
            self.W2 = randn(self.W2_size[0],self.W2_size[1]) *
self.xavier_scale2
```

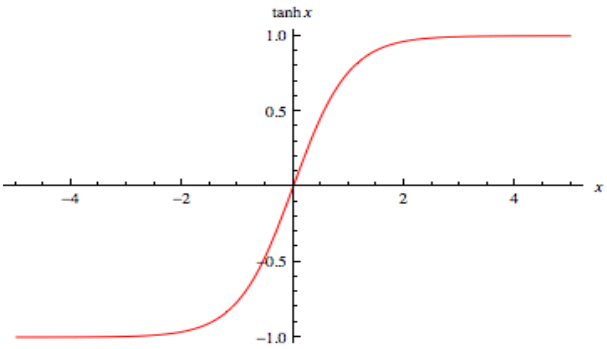
Moreover, the architecture of the network is created by initializing the weights and bias terms.

1.1.4 Activation Unit

Hyperbolic or tanh function is used for activating the all neurons in all layers. The formula for tanh is given by:

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

The following plotting describes the behavior of hyperbolic tangent. Therefore, we can briefly discuss about the hyperbolic tangent. It takes input and scales between -1,1 continuously. It has a derivative that should require when backward propagation. It is similar to sigmoid activation but, the scale of these functions are different. One drawback of it may be vanishing gradient problem since at the near ± 1 point, the derivatives of it approach 0 that may cause problem when traning the model.



The following Python code shows the implementation of the tanh function:

```
    def tanh(self,X):
        return (np.exp(X) - np.exp(-X))/(np.exp(X) + np.exp(-X))

    def tanh_der(self,X):
        return  1-(np.tanh(X)**2)
```

1.1.5 Forward Pass

In forward propagation, we feed input to the network so that get predictions from the model. In the following mathematical expressions, forward propagation is modeled. Note that L_{hidden} is the linear weighted sum of inputs with W_1 then adding bias term B_1. O_{hidden} is the output of the hidden layer. L_{output} is the linear weighted sum of weights W_2 with O_{hidden} then adding bias term B_2. Lastly, O_{output} is the output of the output layer.

$$L_{hidden} = \sum_{i=1}^{N}(W_1 * X + B_1)$$

$$O_{hidden} = \tanh(L_{hidden})$$

$$L_{output} = \sum_{i=1}^{N}(W_2 * O_{hidden} + B_2)$$

$$O_{output} = \tanh(L_{output})$$

The following Python code shows the implementation of the forward propagation:

```
def forward(self,X):

    Z1 = (X @ self.W1)  + self.B1
    A1 = np.tanh(Z1)
    Z2 = (A1 @ self.W2) + self.B2
    A2 = np.tanh(Z2)

    return {"Z1": Z1,"A1": A1,"Z2": Z2,"A2": A2}
```

Note that the formula $\sum_{i=1}^{N}(W_1 * X + B_1)$ represents $\sum_{i=1}^{N}(W_i * X_i + B_i)$ but for the syntax simplicity $\sum_{i=1}^{N}(W_1 * X + B_1)$ will be used along this paper.

1.1.6 Calculation of Loss

For this question, statistical performance measuring functions will be utilized such as mean squared error (MSE) and mean classification error (MCE). The following mathematical expressions describes the behavior of the MSE and MCE function.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(O_{output} - \text{Y})^2$$

Therefore, MSE function takes a prediction generated by the model and the label of training data, compare them by substracting each other and to eliminate negative values MSE function takes to sqaure of the difference. Taking square also helps us to punish outliers in the dataset. Finally, taking the mean of the differences will equate the MSE error.

Moreover, along this question MCE function will be utilized such that we can evaluate the model. We are going to calculate the accuracy of the model by following equation:

$$\text{MCE} = \frac{\#\text{ of correctly classified classes}}{\#\ of\ total\ samples}$$

Hence, statisticallly speaking, accuracy score is general assessing function for model performance. But it can be misleading sometimes, to prevent this F-1 score and Recall functions should be used colloboratively.

Here is the Python implementation of the loss functions:

```
    def Loss(self,pred,y_true):

        mse = np.square(pred-y_true).mean()

        pred[pred>=0]=1
        pred[pred<0]=-1

        mce = (pred == y_true).mean()

        return {'MSE':mse,'MCE':mce}
```

Note that there is a little manipulation on the predictions such that the values of less than 0 will be equal to -1, vice versa.

1.1.7 Backpropagation

In backpropagation part, the effect of the weights W_1 and W_2 and biases B_1, B_2 on the error (i.e., gradients of the W_1, W_2 and B_1, B_2) is calculated via chain rule. The following equations shows the implementation of the backprogation with single hidden layer neural network.

The following gradients should be calculated by:

$$\frac{\partial Error}{\partial W_2} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial W_2}$$

$$\frac{\partial Error}{\partial B_2} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial B_2}$$

$$\frac{\partial Error}{\partial W_1} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden}} * \frac{\partial O_{hidden}}{\partial L_{hidden}} * \frac{\partial L_{hidden}}{\partial W_1}$$

$$\frac{\partial Error}{\partial B_1} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden}} * \frac{\partial O_{hidden}}{\partial L_{hidden}} * \frac{\partial L_{hidden}}{\partial B_1}$$

Necessary sub gradients:

$$\frac{\partial Error}{\partial O_{output}} = \frac{\partial}{\partial L_{hidden}} \frac{1}{N} * \sum_{i=1}^{N} (O_{output} - Y)^2 = O_{output} - Y$$

$$\frac{\partial O_{output}}{\partial L_{output}} = 1 - tanh(L_{output})^2$$

$$\frac{\partial L_{output}}{\partial W_2} = {O_{hidden}}^T$$

$$\frac{\partial L_{output}}{\partial O_{hidden}} = {W_2}^T$$

$$\frac{\partial O_{hidden}}{\partial L_{hidden}} = 1 - tanh(L_{hidden})^2$$

$$\frac{\partial L_{hidden}}{\partial W_1} = X^T$$

Final Gradients:

$$\frac{\partial Error}{\partial W_2} = (O_{output} - Y) * (1 - tanh(L_{output})^2) * ({O_{hidden}}^T)$$

$$\frac{\partial Error}{\partial B_2} = \sum_{i=1}^{N} ((O_{output} - Y) * 1 - tanh(L_{output})^2)$$

$$\frac{\partial Error}{\partial W_1} = (O_{output} - Y) * (1 - tanh(L_{output})^2) * ({W_2}^T) * (1 - tanh(L_{hidden})^2) * X^T$$

$$\frac{\partial Error}{\partial B_1} = \sum_{i=1}^{N} ((O_{output} - Y) * (1 - tanh(L_{output})^2) * ({W_2}^T) * (1 - tanh(L_{hidden})^2))$$

Python implementation of back propagation is shown below:

```
    def backward(self,outs, X, Y):
        m = (self.batch_size)

        Z1 = outs['Z1']
        A1 = outs['A1']
        Z2 = outs['Z2']
        A2 = outs['A2']

        dZ2 = (A2-Y)* self.tanh_der(Z2)
        dW2 = (1/m) * (A1.T @ dZ2)
        dB2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)

        dZ1 = (dZ2 @ self.W2.T) * self.tanh_der(Z1)
        dW1 = (1/m) * (X.T @ dZ1)
        dB1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)
```

```
        return {"dW1": dW1, "dW2": dW2,
                "dB1": dB1, "dB2": dB2}
```

1.1.8 Stochastic Gradient Descent

Stochastic gradient descent on mini batches is performed via the following equations. Note that the expression '-=' means that subtract the argument from itself and equate itself that will be used for future expressions. (i.e., W_2 -= η * $\frac{\partial Error}{\partial W_2}$ *means that* $W_2 := W_2$ - η * $\frac{\partial Error}{\partial W_2}$ *)*

$$W_2 := W_2 - \eta * \frac{\partial Error}{\partial W_2} \qquad W_1 := W_1 - \eta * \frac{\partial Error}{\partial W_1}$$

$$B_2 := B_2 - \eta * \frac{\partial Error}{\partial B_2} \qquad B_1 := B_1 - \eta * \frac{\partial Error}{\partial B_1}$$

The code implementation of stochastic gradient descent is shown below:

```
def SGD(self,grads):
        self.W1 -= self.lr * grads['dW1']
        self.W2 -= self.lr * grads['dW2']
        self.B1 -= self.lr * grads['dB1']
        self.B2 -= self.lr * grads['dB2'
```

Note that for the following SGD update the notation W_2 -= η * $\frac{\partial Error}{\partial W_2}$ is used for $W_2 := W_2$ - η * $\frac{\partial Error}{\partial W_2}$.

1.1.9 Training/Testing

In this part, training loop is implemented. Firstly, I gave random indexes to the input and their labels to feed to the network. Then, I applied forward propagation to calculate loss. After that, I applied backward propagation to find gradients lastly I applied stochastic gradients descent on mini-batches to update to the network parameters. The code implementation is shown below:

```
    def fit(self,X,Y,X_test,y_test,epochs = 300,verbose=True):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """

        m = self.batch_size

        for epoch in range(epochs):
            perm = np.random.permutation(self.sample_size)
```

```
            for i in range(self.n_update):


                batch_start = i * m
                batch_finish = (i+1) * m
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]


                outs = self.forward(X_feed)
                loss = self.Loss(outs['A2'],y_feed)

                outs_test = self.forward(X_test)
                loss_test = self.Loss(outs_test['A2'],y_test)

                grads = self.backward(outs,X_feed,y_feed)
                self.SGD(grads)

            self.mse_train[f"Epoch:{epoch}"] = loss['MSE']
            self.mce_train[f"Epoch:{epoch}"] = loss['MCE']
            self.mse_test[f"Epoch:{epoch}"] = loss_test['MSE']
            self.mce_test[f"Epoch:{epoch}"] = loss_test['MCE']

            if verbose:
                print(f"[{epoch}/{epochs}] ------> Training :MSE:
{loss['MSE']} and MCE: {loss['MCE']}")
                print(f"[{epoch}/{epochs}] ------> Testing :MSE:
{loss_test['MSE']} and MCE: {loss_test['MCE']}"
```

Tranining procedure is given by:

```
initialize = 'Xavier'
input_size = X_train.shape
batch_size = 18
hidden_neurons = 19
epochs = 200

model = TwoLayerNetwork(input_size,batch_size,hidden_neurons,lr=1e-1)


# %%
model.fit(X_train,y_train,X_test,y_test,epochs)


# %%
net_params = model.parameters()
```

```
[0/200] ------> Training :MSE: 1.1447403914922953 and MCE: 0.4
[0/200] ------> Testing :MSE: 0.9442213055596224 and MCE: 0.553
[1/200] ------> Training :MSE: 1.0919065595170654 and MCE: 0.5
[1/200] ------> Testing :MSE: 0.9495583538467921 and MCE: 0.536
[2/200] ------> Training :MSE: 1.000931384502607 and MCE: 0.5
[2/200] ------> Testing :MSE: 0.939129811531358 and MCE: 0.519
[3/200] ------> Training :MSE: 0.8434865422491816 and MCE: 0.6
[3/200] ------> Testing :MSE: 0.87263182865549 and MCE: 0.679
[4/200] ------> Training :MSE: 1.1795274952836219 and MCE: 0.4
[4/200] ------> Testing :MSE: 0.9997345231382598 and MCE: 0.511
[5/200] ------> Training :MSE: 0.8541945573619136 and MCE: 0.7
[5/200] ------> Testing :MSE: 0.8711529607131027 and MCE: 0.719
[6/200] ------> Training :MSE: 0.9340048749183344 and MCE: 0.5
[6/200] ------> Testing :MSE: 0.8947226933240062 and MCE: 0.585
[7/200] ------> Training :MSE: 0.643306936283468 and MCE: 1.0
[7/200] ------> Testing :MSE: 0.9019979917950763 and MCE: 0.603
[8/200] ------> Training :MSE: 0.824591287659963 and MCE: 0.8
[8/200] ------> Testing :MSE: 0.8521148623609716 and MCE: 0.779
[9/200] ------> Training :MSE: 0.7816436014519516 and MCE: 0.7
[9/200] ------> Testing :MSE: 0.8984704840288469 and MCE: 0.566
[10/200] ------> Training :MSE: 0.8145801970724147 and MCE: 0.6
[10/200] ------> Testing :MSE: 0.8811139600388731 and MCE: 0.634
[11/200] ------> Training :MSE: 1.1047308679671806 and MCE: 0.2
[11/200] ------> Testing :MSE: 0.8924772539440781 and MCE: 0.694
[12/200] ------> Training :MSE: 0.8237576129864557 and MCE: 0.7
```

Therefore, the model is evolving and the results will be dissucced in results part.

1.1.10 Hyperparameter Tuning

I implemented grid search algorithm to find best network parameters. I tried the mentioned configurable parameters. The best results are used in the network. Since it is computationally costly, I did not include this part into appendix.

```python
class GridSearch:
    def __init__(self,param_grid):
        self.param_grid = param_grid

    def fit(self,X,Y,X_val,y_val,wrt):

            if wrt == 'Learning Rate':
                acc = []
                for val in self.param_grid[wrt]:
                    temp_model = TwoLayerNetwork(lr = val)
                    temp_model.fit(X,Y,X_val,y_val,epochs = 300, verbose = False)
                    acc.append(np.mean([value for key,value in net_params['Test_MSE'].items()][-150:-1]))
                best =  self.param_grid[wrt][np.argmin(acc)]

                return best

            elif wrt == 'Hidden Neurons':
                acc = []
                for val in self.param_grid[wrt]:
                    temp_model = TwoLayerNetwork(n_neurons = val)
                    temp_model.fit(X,Y,X_val,y_val,epochs = 300, verbose = False)
                    acc.append(np.mean([value for key,value in net_params['Test_MSE'].items()][-150:-1]))
                    best =  self.param_grid[wrt][np.argmin(acc)]

                return best
```

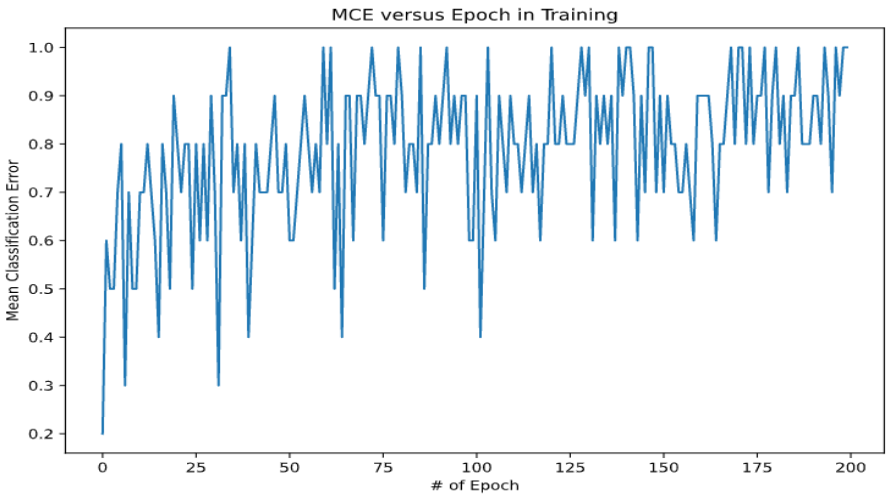
```
            elif wrt == 'Batch Size':
                acc = []
                for val in self.param_grid[wrt]:
                    temp_model = TwoLayerNetwork(batch_size = val)
                    temp_model.fit(X,Y,X_val,y_val,epochs = 300, verbose = False)
                    acc.append(np.mean([value for key,value in net_params['Test_MSE'].items()][-150:-1]))
                    best =  self.param_grid[wrt][np.argmin(acc)]

                return best

            elif wrt ==  'Initializers':
                acc = []
                for val in self.param_grid[wrt]:
                    temp_model = TwoLayerNetwork(distribution = val)
                    temp_model.fit(X,Y,X_val,y_val,epochs = 300, verbose = False)
                    acc.append(np.mean([value for key,value in net_params['Test_MSE'].items()][-150:-1]))
                    best =  self.param_grid[wrt][np.argmin(acc)]

                return best



param_grid = {'Learning Rate' : [1e-2 * i  for i in range(1,11)],
              'Hidden Neurons': [2**i for i in range(4,9)],
              'Batch Size'    : [19 * i for i in range(1,11)],
              'Initializers'  : ['Gauss','Xavier','He']}

grid_search = GridSearch(param_grid = param_grid)
best_lr = grid_search.fit(X_train,y_train,X_test,y_test,wrt = 'Learning Rate')
best_hidden = grid_search.fit(X_train,y_train,X_test,y_test,wrt = 'Hidden Neurons')
best_batch = grid_search.fit(X_train,y_train,X_test,y_test,wrt = 'Batch Size')
best_init = grid_search.fit(X_train,y_train,X_test,y_test,wrt = 'Initializers')
```

Therefore, final values calculated as a:

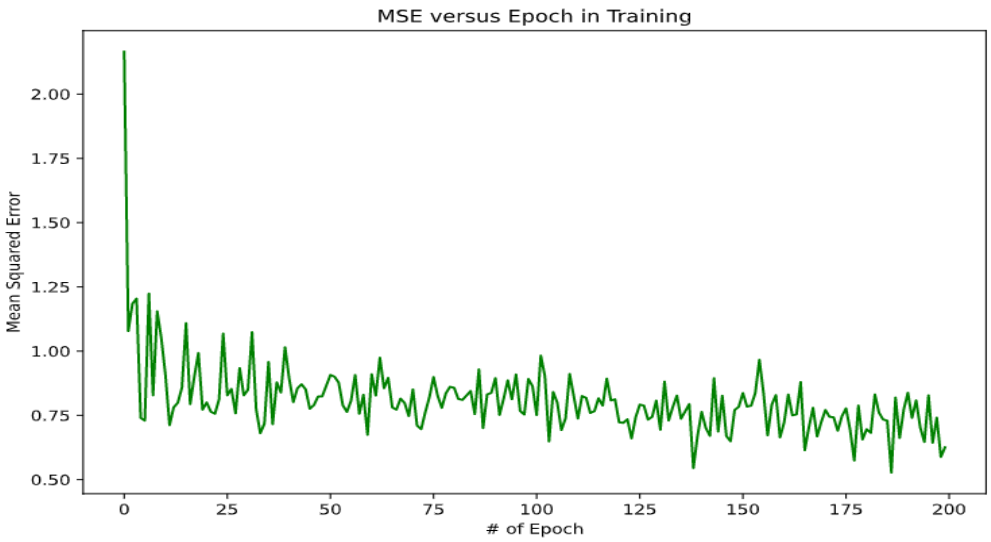
Hyperparameters	Results
Hidden layer number *N*	19
Initialization method	Xavier Initialization
Epochs	200
Batch size	18
Learning rate	1e-1

1.1.11 Result/Discuss

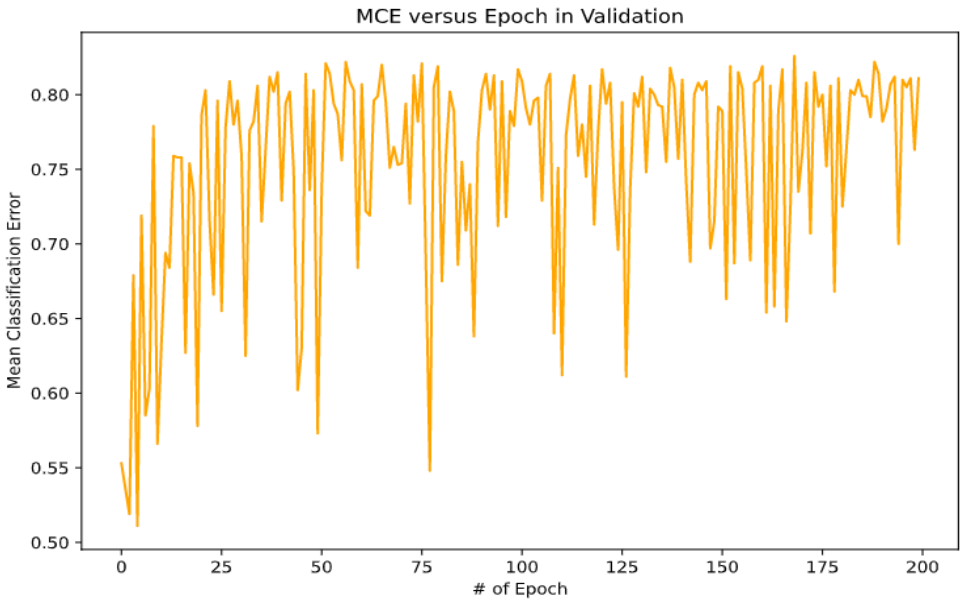
In this part, I am going to plot the learning curves as a function of epoch number for training squared error, testing squared error, training classification error, and testing classsification error. Here are the results:



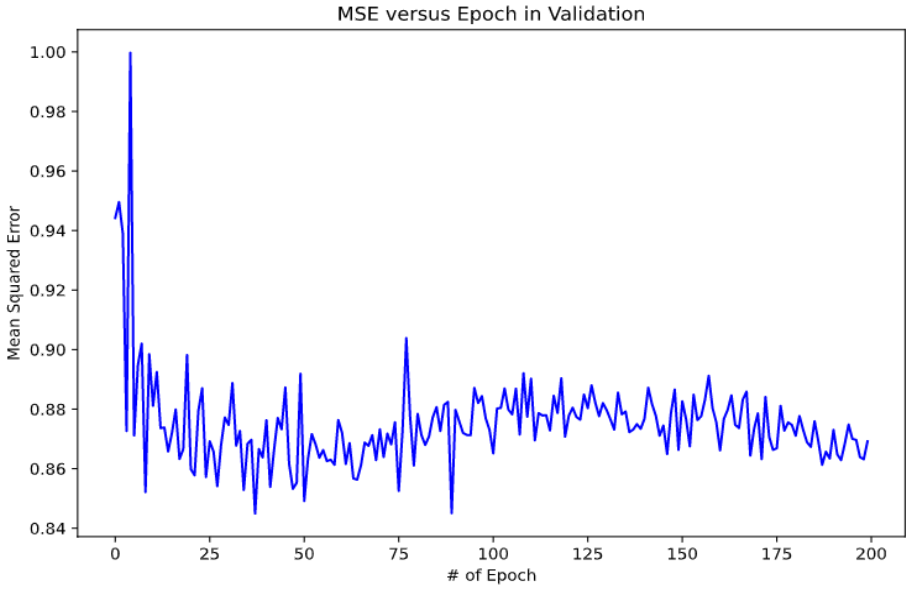
As we expect, MCE is incrased gradually over traning and approaces to the 100% accuracy with little fluctuations. The reason behind this fluctuating are random process while training,i.e., we randomly select to the inputs and feed to the network on mini-batches and my batch size which is 19. Since stochastic gradient descent on mini batches place in the middle of on-line traning and batch traning, these gradually descreased fluctuations are expected. If one wants to eliminate this fluctuations, can try higher batch sizes. In anyway, our model is learned well with approximately 100% accuracy.



On the other hand, we are expecting that MSE loss should decrease gradually while traning. The above figure shows the learning curve of the model so that our MSE cost is decreasing.



On the other hand, while traning,i.e., while our model learning the input-outputs maps, it is important to generalize so that the model performs well on testing data. Hence, in the above figure we see that our tranining accuracy approximately reached 83-85% that is not bad. The reason behind the difference between traning and testing accuracy is variance in the data. We expect that our model should perform well in testing dataset with the certain accuracy that should be near to traning accuracy. In this case, we can say that our model is over fitting a bit, but overall performance is well anyway.



Finally, since the performance of the model is increases gradually, MSE error on testing data should be descreased gradually. Hence, it is decreasing step by step so that our model performed well on the validation data.

1.2 PART B

In this part, I am going to answer to the questions:

- How the squared-error and classifcation error metrics evolve over epochs for the training versus the testing sets? (Answered in above part)

- Is squared error an adequate predictor of classification error?

1.2.1 Is squared error an adequate predictor of classification error?

Sqaured errors such as RMSE(root mean square error), MSE (mean sqaure error) and Ṛ^2 errors is widely used in regression tasks not for the classification tasks. (i.e., they can be used, but there are more optimal loss functions) There are several reasons behind this. Firstly, mean sqaured errors means that we assumed that underlying data has been generated by gauss distribution, in bayesian terms, this means that gaussian prior. In real classification tasks, this may not be the case. For example, if we are planning to use maximum likelihood estimation when assuming data is drawn for gauss distribution that is actually wrong assumption, we get mean sqaured error as a loss function for optimization of the model. The following mathematical expressions are the proof of that.

Each target value Y can be defined by gauss distribution.

$$\mathrm{P}(Y\,|X) = \mathrm{N}(Y;\ \hat{\mathrm{Y}}, \sigma^2)$$

Using maximum likelihood estimation, the multiplication of gauss distributions for each target value:

$$\mathrm{Ł}\,(\,Y^{(1)}, Y^{(2)}, \dots, Y^{(m)} |\, \mathrm{X}^{(1)}, \mathrm{X}^{(2)}, \dots, \mathrm{X}^{(m)}\,) = \prod_{i=1}^{m} \mathrm{P}\big(Y^{(i)}\,\big|\mathrm{X}^{(i)}\big)$$

$$= \prod_{i=1}^{m} \mathrm{N}(Y^{(i)}; \mu = \hat{\mathrm{Y}}^{(i)}, \sigma^2)$$

$$= \prod_{i=1}^{m} \sqrt{\frac{1}{2\pi\sigma^2}} * e^{-\frac{-\left(Y^{(i)} - \hat{\mathrm{Y}}^{(i)}\right)^2}{2\sigma^2}}$$

To simplification, we can take natural log of the likelihood function:

$$\log(\text{Ł}(Y^{(1)}, Y^{(2)}, \dots, Y^{(m)} | \text{X}^{(1)}, \text{X}^{(2)}, \dots, \text{X}^{(m)}) = \log \prod_{i=1}^{m} \sqrt{\frac{1}{2\pi\sigma^2}} * e^{-\frac{-\left(Y^{(i)} - \hat{\text{Y}}^{(i)}\right)^2}{2\sigma^2}}$$

$$= \sum_{i=1}^{m} \log\left(\sqrt{\frac{1}{2\pi\sigma^2}}\right) - \frac{-\left(Y^{(i)} - \hat{\text{Y}}^{(i)}\right)^2}{2\sigma^2}$$

$$= \sum_{i=1}^{m} -\frac{1}{2\sigma^2} \left(Y^{(i)} - \hat{\text{Y}}^{(i)}\right)^2$$

Since we know that maximizing a function is the same as minimizing the negative of that function:

$$-\log(\text{Ł}(Y^{(1)}, Y^{(2)}, \dots, Y^{(m)} | \text{X}^{(1)}, \text{X}^{(2)}, \dots, \text{X}^{(m)}) = \sum_{i=1}^{m} \frac{1}{2\sigma^2} \left(Y^{(i)} - \hat{\text{Y}}^{(i)}\right)^2$$

Since variance σ^2 is not depend on $\hat{\text{Y}}^{(i)}$, we can simply ignore it:

$$-\log(\text{Ł}(Y^{(1)}, Y^{(2)}, \dots, Y^{(m)} | \text{X}^{(1)}, \text{X}^{(2)}, \dots, \text{X}^{(m)}) = \sum_{i=1}^{m} \left(Y^{(i)} - \hat{\text{Y}}^{(i)}\right)^2$$

Finally, we can scale the negative log likelihood function by $\frac{1}{m}$ that is the number of traning sample. We get mean sqaured error:

$$\text{MSE} = -\frac{1}{m} \log(\text{Ł}(Y^{(1)}, Y^{(2)}, \dots, Y^{(m)} | \text{X}^{(1)}, \text{X}^{(2)}, \dots, \text{X}^{(m)})$$

$$= \frac{1}{2m} * \sum_{i=1}^{m} \left(Y^{(i)} - \hat{\text{Y}}^{(i)}\right)^2$$

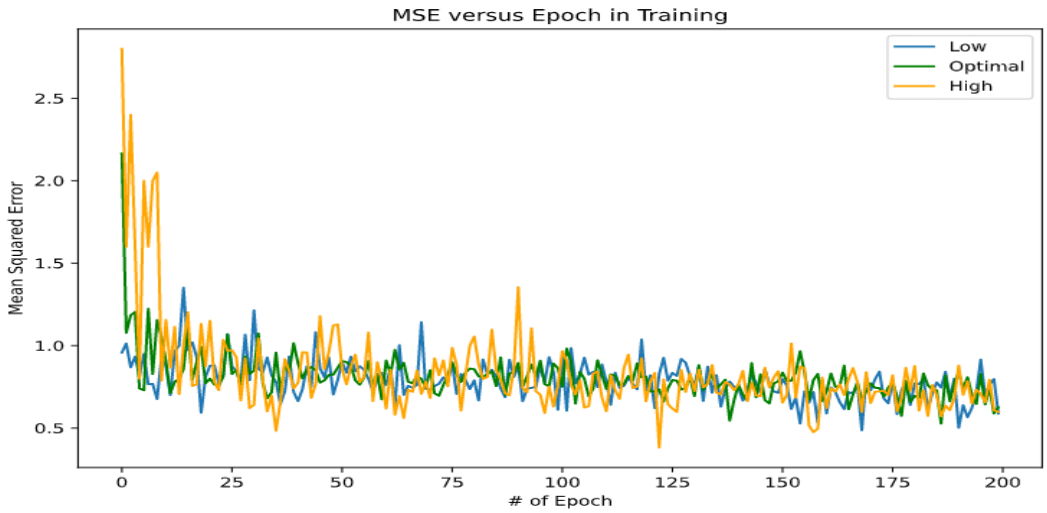
Secondly, MSE loss is non-convex for our task, i.e, binary classification task. Hence, it is not quaranteed to minimize the loss for binary classification problems. The reason behind this is the nature of the mean sqaured error loss, i.e., mse takes real-valued values from $-\infty \; to \; \infty$. In our case, model predicts 1 or -1. Further, e.q., if weights and biases initialized large values, the concave part of the MSE gradient will not work properly that will cause problem to the model while traning. Hence, MSE may not be the perfect loss function for classification tasks since it's nature, but it is a powerful metric for regression model such as linear regression. For our classification task, binary cross-entropy is a well decided choice. Finally, even MSE loss function shows the learning curve and decreases gradually while tranining, it is not the best choice for classification tasks.

1.3 PART C

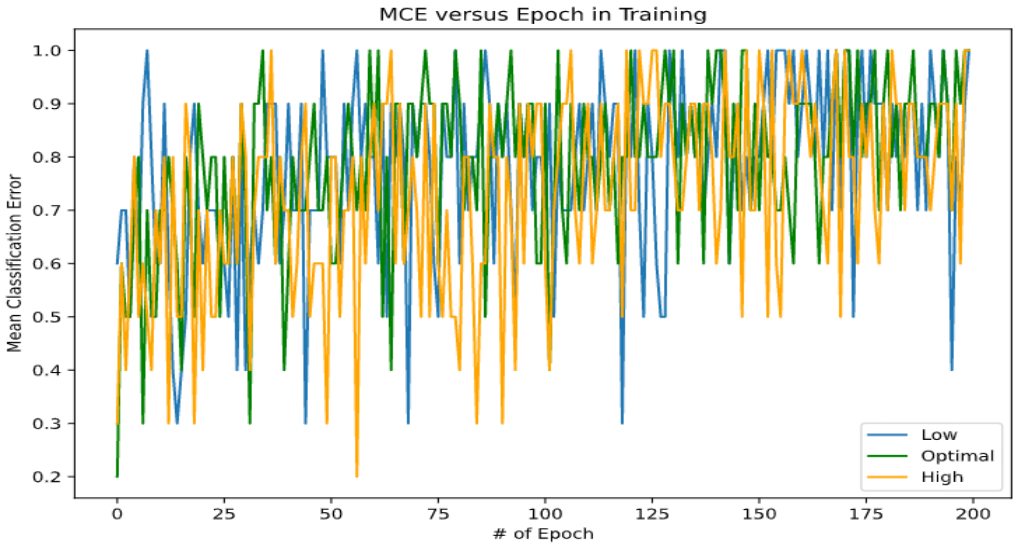
In this part, the effects of the hidden layer neuron will be discussed based on the conclusions drawn from the plotting's generated by training separate neural networks with substantially higher and lower values of hidden neurons.

The number of hidden layer units is one of the configurable parameters in the network that should be optimized by several techniques. In this question, I tried 2 hidden layer units, that are substantially lower and higher respectively, 8 and 152 to test the network results. Here are results:

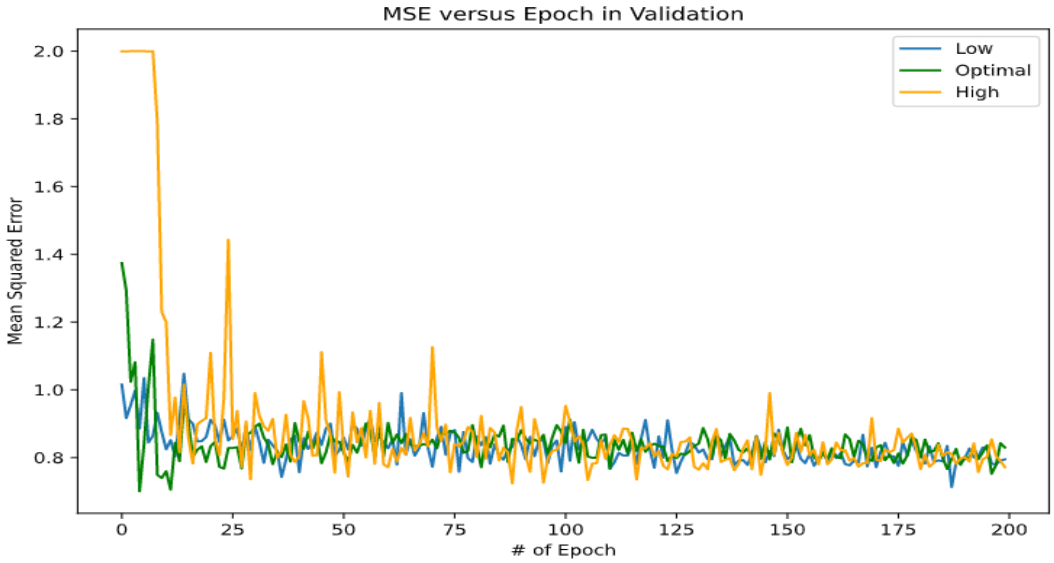
```
low_hidden_model = TwoLayerNetwork(input_size,batch_size,n_neurons=8,
lr=1e-1)
high_hidden_model = TwoLayerNetwork(input_size,batch_size,n_neurons=152,
lr=1e-1)

low_hidden_model.fit(X_train,y_train,X_test,y_test,epochs = 200, verbose
= False)
high_hidden_model.fit(X_train,y_train,X_test,y_test,epochs =200, verbose
= False)

low_hidden_params = low_hidden_model.parameters()
high_hidden_params = high_hidden_model.parameters()
```



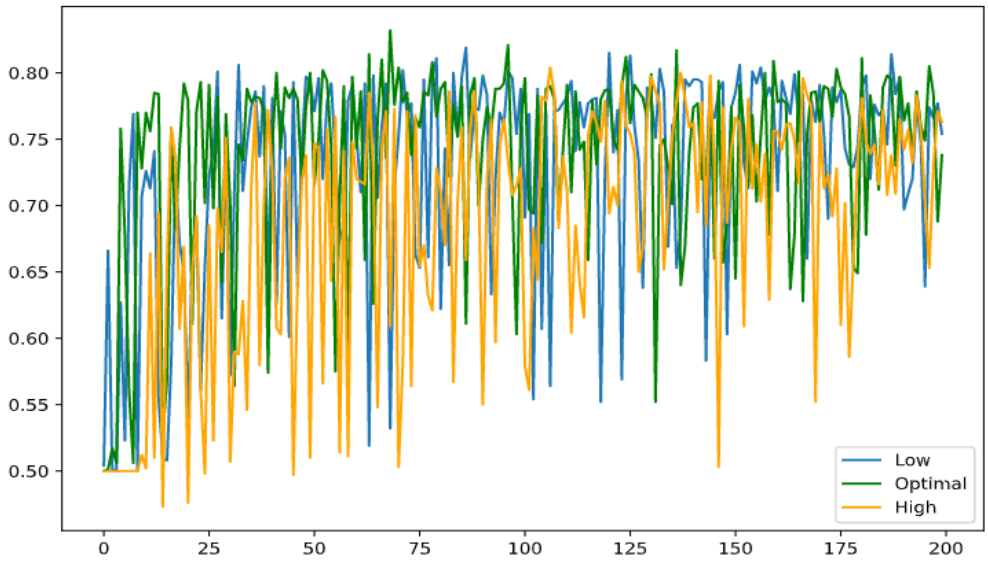
The mean squared error is decreased in all neuron sizes as expected. The model with low hidden neuron size performed well also with more fluctuations with respect to the optimal one. Furthermore, the model with high hidden layer size is converged also but with more fluctuations. However, overall results satisfactory.



In the case of mean classification error, all of the models are reached the 100% training accuracy. However, the model with low hidden unit size has more fluctuations w.r.t optimal one. Also, the model with high hidden neuron size, performed well but has huge fluctuations and unbalanced errors while training. Also, since the matrix size is increased, it is computationally costly.



MSE errors due different hidden unit size in validation set is plotted above. We see that all converges approximately global minima. However, the model with high hidden unit performed more fluctuations while the model with low hidden unit has less fluctuations.



Finally, our models with different hidden layer units performed satisfactory in validation data. The optimal one reached approximately ~83-87% accuracy while others remains approximately 80%.

1.4 PART D

In this part, the task is to design and train a separate network with two hidden layers. The following parts, only changes will be discussed to prevent repetitions.

1.4.1 Architecture

Input layer has a dimension $\mathbb{R}^{1024}$, even the hidden layer H_1 and H_2 neuron unit is a hyperparameter, I tune the neurons so that result will be $H_1 \in \mathbb{R}^{76}$ *and* $H_2 \in \mathbb{R}^{380}$ that we we will see in the following parts and lastly output layer $\in \mathbb{R}^1$ since the task is binary classification cats versus cars. The following code initializes the weights and biases.

```
class ThreeLayerNetwork:

    def __init__(self,input_size,batch_size,h1_neurons,h2_neurons, mean =
0,std = 1,lr =1e-1,distribution = 'Xavier'):
        np.random.seed(15)
        self.lr = lr
        self.mse_train = {}
        self.mce_train = {}
        self.mse_test = {}
        self.mce_test = {}
        self.prev_updates = {'W1':0,'W2':0,'W3':0,
                             'B1':0,'B2':0,'B3':0}

        self.h1_neurons = h1_neurons
        self.h2_neurons = h2_neurons

        self.sample_size = input_size[0]
        self.feature_size = input_size[1]
```

```
        self.batch_size = batch_size
        self.mean,self.std = mean,std

        self.dist = distribution


        self.n_update = round((self.sample_size/self.batch_size))

        self.W1_size = self.feature_size,self.h1_neurons
        self.W2_size = self.h1_neurons,self.h2_neurons
        self.W3_size = self.h2_neurons,1

        self.B1_size = 1,h1_neurons
        self.B2_size = 1,h2_neurons
        self.B3_size = 1,1




        if (self.dist == 'Zero') :
            self.W1 = np.zeros((self.W1_size))
            self.W2 = np.zeros((self.W2_size))
            self.W3 = np.zeros((self.W3_size))
            self.B1 = np.zeros((self.B1_size))
            self.B2 = np.zeros((self.B2_size))
            self.B3 = np.zeros((self.B3_size))

        elif (self.dist == 'Gauss'):
            self.W1 = Gauss(loc = self.mean, scale = self.std, size =
(self.W1_size)) * 0.01
            self.W2 = Gauss(loc = self.mean, scale = self.std, size =
(self.W2_size)) * 0.01
            self.W3 = Gauss(loc = self.mean, scale = self.std, size =
(self.W3_size)) * 0.01

            self.B1 = Gauss(loc = self.mean, scale = self.std, size =
(self.B1_size)) * 0.01
            self.B2 = Gauss(loc = self.mean, scale = self.std, size =
(self.B2_size)) * 0.01
            self.B3 = Gauss(loc = self.mean, scale = self.std, size =
(self.B3_size)) * 0.01

        elif (self.dist == 'He'):
            self.he_scale1 = np.sqrt(2/self.feature_size)
            self.he_scale2 = np.sqrt(2/self.h1_neurons)
            self.he_scale3 = np.sqrt(2/self.h2_neurons)

            self.W1 = randn(self.W1_size[0],self.W1_size[1]) *
self.he_scale1
            self.W2 = randn(self.W2_size[0],self.W2_size[1]) *
self.he_scale2
            self.W3 = randn(self.W2_size[0],self.W2_size[1]) *
self.he_scale3

            self.B1 = randn(self.B1_size[0],self.B1_size[1]) *
self.he_scale1
            self.B2 = randn(self.B2_size[0],self.B2_size[1]) *
self.he_scale2
            self.B3 = randn(self.B3_size[0],self.B3_size[1]) *
self.he_scale3
```

```
        elif (self.dist == 'Xavier'):
            self.xavier_scale1 =
np.sqrt(2/(self.feature size+self.h1 neurons))
            self.xavier_scale2 =
np.sqrt(2/(self.h1_neurons+self.h2_neurons))
            self.xavier_scale3 = np.sqrt(2/(self.h2_neurons+1))

            self.W1 = randn(self.W1_size[0],self.W1_size[1]) *
self.xavier_scale1
            self.W2 = randn(self.W2_size[0],self.W2_size[1]) *
self.xavier_scale2
            self.W3 = randn(self.W3_size[0],self.W3_size[1]) *
self.xavier_scale3

            self.B1 = randn(self.B1_size[0],self.B1_size[1]) *
self.xavier_scale1
            self.B2 = randn(self.B2_size[0],self.B2_size[1]) *
self.xavier_scale2
            self.B3 = randn(self.B3_size[0],self.B3_size[1]) *
self.xavier_scale3
```

*1.4.2 **Forward Pass***

As we did before, hyperbolic or tanh function is used for activating the all neurons in all layers.

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

The forward pass is processed firstly by inputting X to the network. Weighted sum of the input matrix with W_1 is added B_1 to feed the first activation in the input that is O_{hidden_1} . Then, O_{hidden_1} is multiplied by W_2 and added B_2 to feed the 2nd activation. Lastly, O_{hidden_2} is multiplied by the W_3 then added B_3 to feed the output layer's activation to make prediction. The following mathematical expressions explains the process.

$$L_{hidden_1} = \sum_{i=1}^{N}(W_1 * X + B_1)$$

$$O_{hidden_1} = \tanh(L_{hidden_1})$$

$$L_{hidden_2} = \sum_{i=1}^{N}(W_2 * O_{hidden_1} + B_2)$$

$$O_{hidden_2} = \tanh(L_{hidden_2})$$

$$L_{output} = \sum_{i=1}^{N}(W_3 * O_{hidden_2} + B_3)$$

$$O_{output} = \tanh(L_{hidden_2})$$

The implementation of the forward propagation is shown below:

```
def forward(self,X):

    Z1 = (X @ self.W1)  + self.B1
    A1 = np.tanh(Z1)
    Z2 = (A1 @ self.W2) + self.B2
    A2 = np.tanh(Z2)
    Z3 = (A2 @ self.W3) + self.B3
    A3 = np.tanh(Z3)

    return {"Z1": Z1,"A1": A1,
            "Z2": Z2,"A2": A2,
            "Z3": Z3,"A3": A3}
```

1.4.3 Calculation of Loss

In this part, we are using same code and same expressions so that no need to repeat same procedures.

1.4.4 Backpropagation

In this part, all gradients w.r.t. loss function is calculated both analytically and in Python.

Necessary Gradients

$$\frac{\partial Error}{\partial W_3} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial W_3}$$

$$\frac{\partial Error}{\partial B_3} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial B_3}$$

$$\frac{\partial Error}{\partial W_2} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden_2}} * \frac{\partial O_{hidden_2}}{\partial L_{hidden_2}} * \frac{\partial L_{hidden_2}}{\partial W_2}$$

$$\frac{\partial Error}{\partial B_2} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden}} * \frac{\partial O_{hidden}}{\partial L_{hidden}} * \frac{\partial L_{hidden}}{\partial B_2}$$

$$\frac{\partial Error}{\partial W_1} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden_2}} * \frac{\partial O_{hidden_2}}{\partial L_{hidden_2}} * \frac{\partial L_{hidden_2}}{\partial O_{hidden_1}} * \frac{\partial O_{hidden_1}}{\partial L_{hidden_1}} * \frac{\partial L_{hidden_1}}{\partial W_1}$$

$$\frac{\partial Error}{\partial B_1} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden_2}} * \frac{\partial O_{hidden_2}}{\partial L_{hidden_2}} * \frac{\partial L_{hidden_2}}{\partial O_{hidden_1}} * \frac{\partial O_{hidden_1}}{\partial L_{hidden_1}} * \frac{\partial L_{hidden_1}}{\partial B_1}$$

Final gradients:

$$\frac{\partial Error}{\partial W_3} = (O_{output} - Y) * (1 - tanh(L_{output})^2) * ({O_{hidden_2}}^T)$$

$$\frac{\partial Error}{\partial B_3} = \sum_{i=1}^{N} ((O_{output} - Y) * 1 - tanh(L_{output})^2)$$

$$\frac{\partial Error}{\partial W_2} = (O_{output} - Y) * (1 - tanh(L_{output})^2) * ({W_3}^T) * (1 - tanh(L_{hidden_2})^2) * {O_{hidden_1}}^T$$

$$\frac{\partial Error}{\partial B_2} = \sum_{i=1}^{N} ((O_{output} - Y) * (1 - tanh(L_{output})^2) * ({W_3}^T) * (1 - tanh(L_{hidden_2})^2))$$

$$\frac{\partial Error}{\partial W_1} = (O_{output} - Y) * (1 - tanh(L_{output})^2) * ({W_3}^T) * (1 - tanh(L_{hidden_2})^2) * ({W_2}^T) * (1 - tanh(L_{hidden_1})^2) * (X^T)$$

$$\frac{\partial Error}{\partial B_1} = \sum_{i=1}^{N} ((O_{output} - Y) * (1 - tanh(L_{output})^2) * ({W_3}^T) * (1 - tanh(L_{hidden_2})^2) * ({W_2}^T) * (1 - tanh(L_{hidden_1})^2))$$

The following code is the implementation of back propagation:

```
def backward(self,outs, X, Y):
    m = self.batch_size

    Z1 = outs['Z1']
    A1 = outs['A1']
    Z2 = outs['Z2']
    A2 = outs['A2']
    Z3 = outs['Z3']
    A3 = outs['A3']


    dZ3 = (A3-Y)* self.tanh_der(Z3)
    dW3 = (1/m) * (A2.T @ dZ3)
    dB3 = (1/m) * np.sum(dZ3, axis=0, keepdims=True)

    dZ2 = np.multiply((dZ3 @ self.W3.T), self.tanh_der(Z2))
    dW2 = (1/m) * (A1.T @ dZ2)
    dB2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)


    dZ1 = np.multiply((dZ2 @ self.W2.T), self.tanh_der(Z1))
    dW1 = (1/m) * (X.T @ dZ1)
    dB1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)


    return {"dW1": dW1, "dW2": dW2,"dW3": dW3,
            "dB1": dB1, "dB2": dB2,"dB3": dB3}
```

1.4.5 Stochastic Gradient Descent

Stochastic gradient descent on mini-batches is implemented in following expression:

$$W_3 -= \eta * \frac{\partial Error}{\partial W_3} \qquad W_2 -= \eta * \frac{\partial Error}{\partial W_2}$$

$$B_3 -= \eta * \frac{\partial Error}{\partial B_3} \qquad B_2 -= \eta * \frac{\partial Error}{\partial B_2}$$

$W_1 -= \eta * \frac{\partial Error}{\partial W_1}$ $B_1 -= \eta * \frac{\partial Error}{\partial B_1}$

The Python implementation is shown below:

```
def SGD(self,grads,momentum = False, mom_coeff = None):
        if momentum:
            self.W1 += (-self.lr * grads['dW1'] +  mom_coeff *
self.prev_updates['W1'])
            self.W2 += (-self.lr * grads['dW2'] +  mom_coeff *
self.prev_updates['W2'])
            self.W3 += (-self.lr * grads['dW3'] +  mom_coeff *
self.prev_updates['W3'])
            self.B1 += (-self.lr * grads['dB1'] +  mom_coeff *
self.prev_updates['B1'])
            self.B2 += (-self.lr * grads['dB2'] +  mom_coeff *
self.prev_updates['B2'])
            self.B3 += (-self.lr * grads['dB3'] +  mom_coeff *
self.prev_updates['B3'])

            self.prev_updates['W1'] = - self.lr * grads['dW1'] +
mom_coeff * self.prev_updates['W1']
            self.prev_updates['W2'] = - self.lr * grads['dW2'] +
mom_coeff * self.prev_updates['W2']
            self.prev_updates['W3'] = - self.lr * grads['dW3'] +
mom_coeff * self.prev_updates['W3']
            self.prev_updates['B1'] = - self.lr * grads['dB1'] +
mom_coeff * self.prev_updates['B1']
            self.prev_updates['B2'] = - self.lr * grads['dB2'] +
mom coeff * self.prev updates['B2']
            self.prev_updates['B3'] = - self.lr * grads['dB3'] +
mom_coeff * self.prev_updates['B3']

        else:
            self.W1 -= self.lr * grads['dW1']
            self.W2 -= self.lr * grads['dW2']
            self.W3 -= self.lr * grads['dW3']
            self.B1 -= self.lr * grads['dB1']
            self.B2 -= self.lr * grads['dB2']
            self.B3 -= self.lr * grads['dB3']
```

Note that this code will be used also in part (e). We are interested in the else condition in the figure.

1.4.6 Training/Testing

The same traning loop is implemented, the difference between the 1st part is just we added one more hidden layer. The code is given below:

```
def fit(self,X,Y,X_test,y_test,epochs,momentum = False, mom_coeff =
None,verbose=True):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """
```

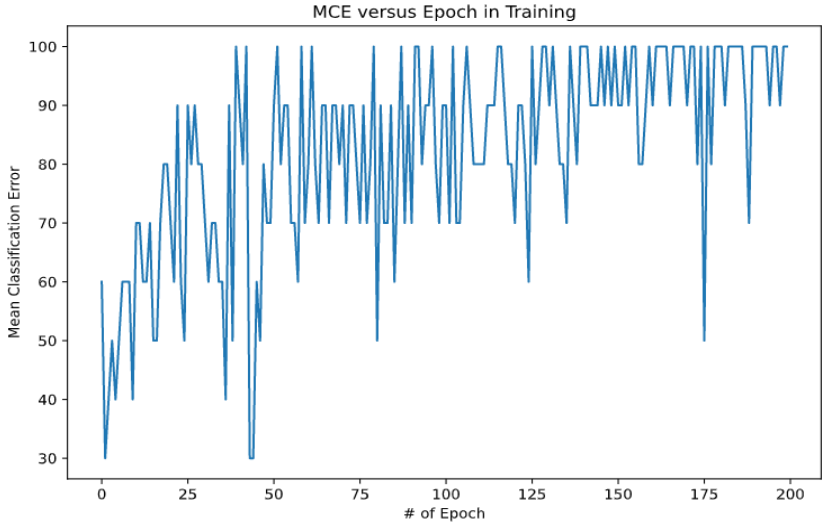
```
        for epoch in range(epochs):
            perm = np.random.permutation(self.sample size)
            print(f"Epoch num: {epoch}")
            for idx in range(self.n_update):


                batch_start = idx * self.batch_size
                batch_finish =(idx+1) * self.batch_size
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]


                outs = self.forward(X_feed)
                loss = self.Loss(outs['A3'],y_feed)

                outs_test = self.forward(X_test)
                loss_test = self.Loss(outs_test['A3'],y_test)

                grads = self.backward(outs,X_feed,y_feed)
                self.SGD(grads, momentum = momentum, mom_coeff =
mom_coeff)

            self.mse_train[f"Epoch:{epoch}"] = loss['MSE']
            self.mce_train[f"Epoch:{epoch}"] = loss['MCE']
            self.mse_test[f"Epoch:{epoch}"] = loss_test['MSE']
            self.mce_test[f"Epoch:{epoch}"] = loss_test['MCE']

            if verbose:
                print(f"[{epoch}/{epochs}] ------> Training :MSE:
{loss['MSE']} and MCE: {loss['MCE']}")
                print(f"[{epoch}/{epochs}] ------> Testing  :MSE:
{loss_test['MSE']} and MCE: {loss_test['MCE']}")
```

1.4.7 Result/Discuss

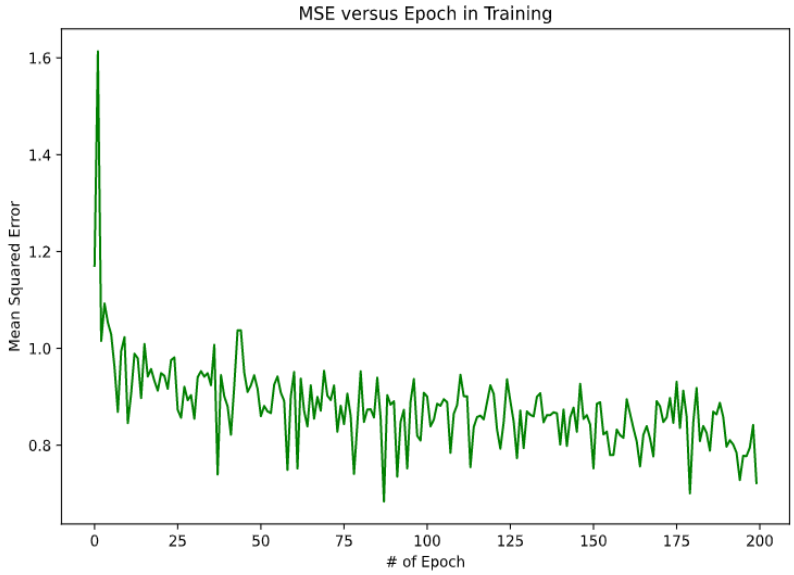
The discuss the results, I plotted learning curve as a function of epoch number. Here are the results:

Hyperparameters	Results
1st Hidden layer number N_1	76
2nd Hidden layer number N_2	380
Initialization method	Xavier Initialization
Epochs	200
Batch size	18
Learning rate	1e-1

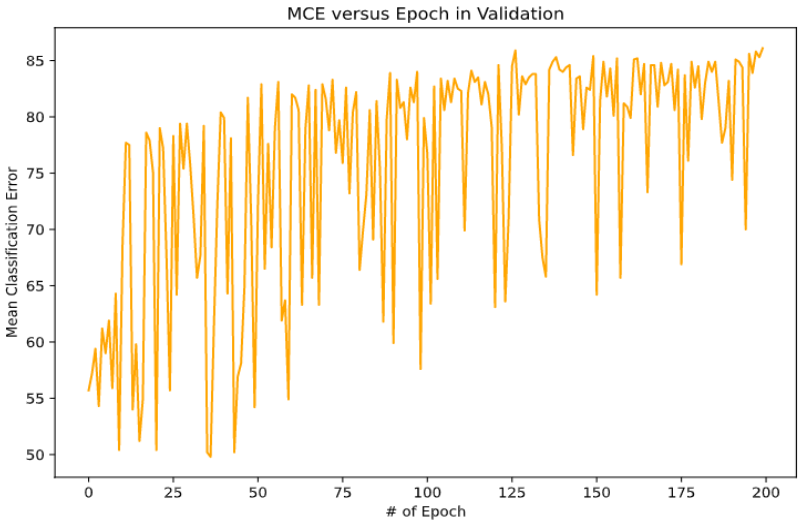
These results are found by grid search. The function created in the previous part is used in this part also. I did not include the same code to avoid repeating myself. The plottings are shown in below.



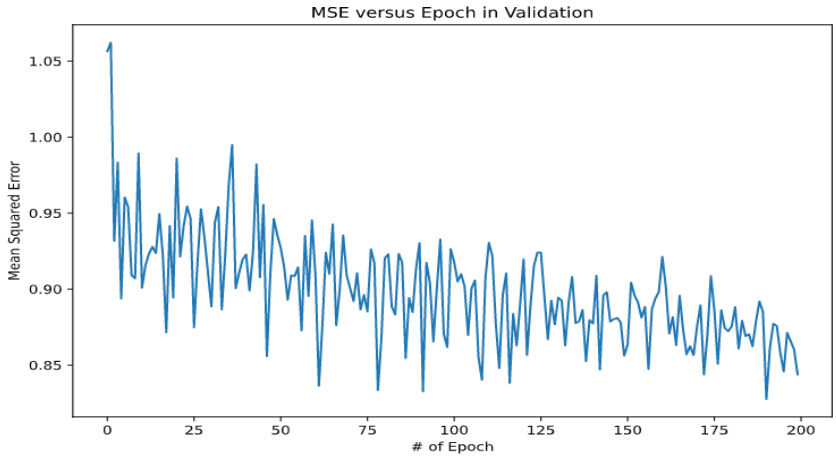
We see that with the increment of the epoch number, our accuracy score is increased gradally up to 100% . Hence, our model with 2 hidden layer is performed so well on tranining data. When we compare the results betwen previous model, we see that this model is more stable(i.e., fast convergence, less oscillatory) after 200 epochs w.r.t. previous network.



As expected, MSE loss is decrased gradully with the epoch number, and there are no big fluctuations hence this learning curve is more stable and robust w.r.t. previous model.



While assessing model performance, the 'generalization' is crucial and we see that our model with 2 hidden layer perfomed well on testing data with the accuracy of apporixamately 87%. The previous network perfomed on validation data in accuracy of maximum 85%. Therefore, we can say that our model is outperformed w.r.t previous model. Note that I want to keep my epochs in 200 to be consistent with previous parts. But, in this model my model may converge 90% accuracy with the increase of epoch number.



MSE loss is decreased as expected on validation data step by step with the epoch number and it is more stable than the previous network.

Therefore, the model with 2 hidden layer outperformed the model with single hidden layer.

1.5 PART E

In this part, stochastic gradient descent should be implemented with the momentum where momentum coefficient is α. The mathematical expression of momentum SGD is explained

below where $\Delta W_i{}^N$ and $\Delta B_i{}^N$ corresponds to update element currently and $\Delta W_i{}^{N-1} and$ $\Delta B_i{}^{N-1}$ corresponds to the previous updates for i = 1,2,3.

Momentum Updates

$$\Delta W_3{}^N = -\eta * \frac{\partial Error}{\partial W_3} + \alpha * \Delta W_3{}^{N-1}$$

$$W_3 \mathrel{+}= \Delta W_3{}^N$$

$$\Delta B_3{}^N = -\eta * \frac{\partial Error}{\partial B_3} + \alpha * \Delta B_3{}^{N-1}$$

$$B_3 \mathrel{+}= \Delta B_3{}^N$$

$$\Delta W_2{}^N = -\eta * \frac{\partial Error}{\partial W_2} + \alpha * \Delta W_2{}^{N-1}$$

$$W_2 \mathrel{+}= \Delta W_2{}^N$$

$$\Delta B_2{}^N = -\eta * \frac{\partial Error}{\partial B_2} + \alpha * \Delta B_2{}^{N-1}$$

$$B_2 \mathrel{+}= \Delta B_2{}^N$$

$$\Delta W_1{}^N = -\eta * \frac{\partial Error}{\partial W_1} + \alpha * \Delta W_1{}^{N-1}$$

$$W_1 \mathrel{+}= \Delta W_1{}^N$$

$$\Delta B_1{}^N = -\eta * \frac{\partial Error}{\partial B_1} + \alpha * \Delta B_1{}^{N-1}$$

$$B_1 \mathrel{+}= \eta * \frac{\partial Error}{\partial B_1}$$

The Python implementation is shown below: (Note that same piece of code is in above, but I added here just for sake of simplicity)

```
    def SGD(self,grads,momentum = False, mom_coeff = None):
        if momentum:
            self.W1 += (-self.lr * grads['dW1'] +  mom_coeff *
self.prev_updates['W1'])
            self.W2 += (-self.lr * grads['dW2'] +  mom_coeff *
self.prev_updates['W2'])
            self.W3 += (-self.lr * grads['dW3'] +  mom_coeff *
self.prev_updates['W3'])
            self.B1 += (-self.lr * grads['dB1'] +  mom coeff *
self.prev_updates['B1'])
            self.B2 += (-self.lr * grads['dB2'] +  mom_coeff *
self.prev_updates['B2'])
            self.B3 += (-self.lr * grads['dB3'] +  mom_coeff *
self.prev_updates['B3'])

            self.prev_updates['W1'] = - self.lr * grads['dW1'] +
mom_coeff * self.prev_updates['W1']
            self.prev_updates['W2'] = - self.lr * grads['dW2'] +
mom_coeff * self.prev_updates['W2']
            self.prev_updates['W3'] = - self.lr * grads['dW3'] +
mom_coeff * self.prev_updates['W3']
            self.prev_updates['B1'] = - self.lr * grads['dB1'] +
mom_coeff * self.prev_updates['B1']
            self.prev_updates['B2'] = - self.lr * grads['dB2'] +
mom_coeff * self.prev_updates['B2']
            self.prev_updates['B3'] = - self.lr * grads['dB3'] +
mom_coeff * self.prev_updates['B3']
```

1.5.1 Results/Discuss

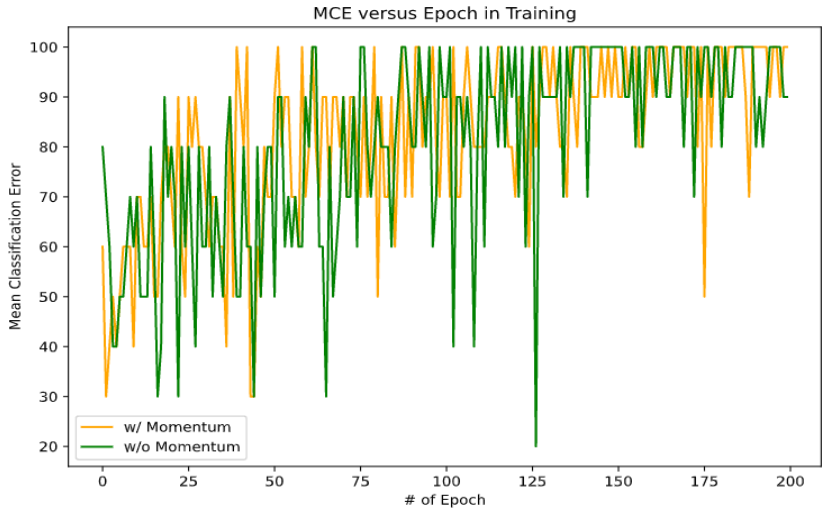
The following code trained the network with momentum and then the following figures are plotted to discuss the results. Before that, my hyperparamaters table is:

Hyperparameters	Results
1st Hidden layer number N_1	76
2nd Hidden layer number N_2	380
Initialization method	Xavier Initialization
Epochs	200
Batch size	18
Learning rate	1e-1
Momentum coefficient	1e-1

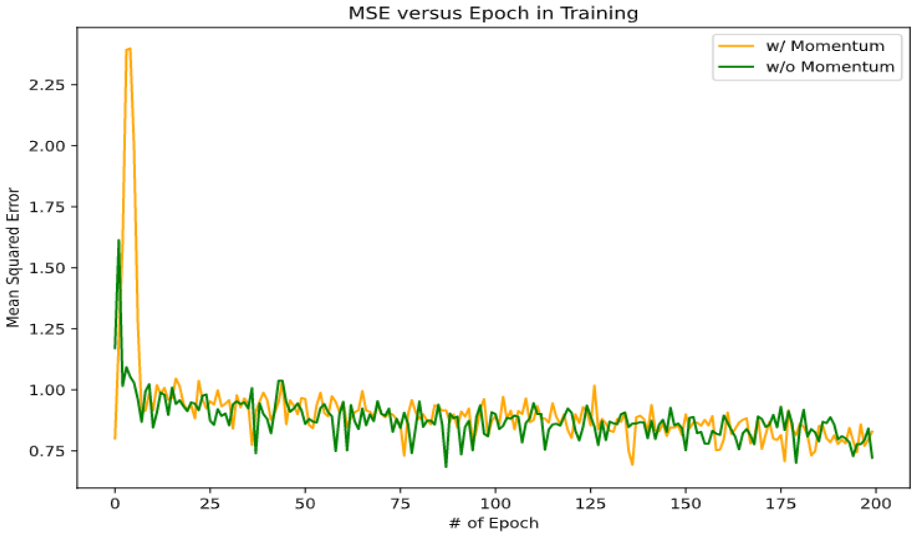
```
with_mom_model =
ThreeLayerNetwork(input_size,batch_size,h1_neurons,h2_neurons)


with_mom_model.fit(X_train,y_train,X_test,y_test,epochs, momentum = True,
mom_coeff = 0.1)


net_params3 = with_mom_model.parameters()
```

In general, momentum stochastic gradient descent provides 2 certain advantages over classical one:

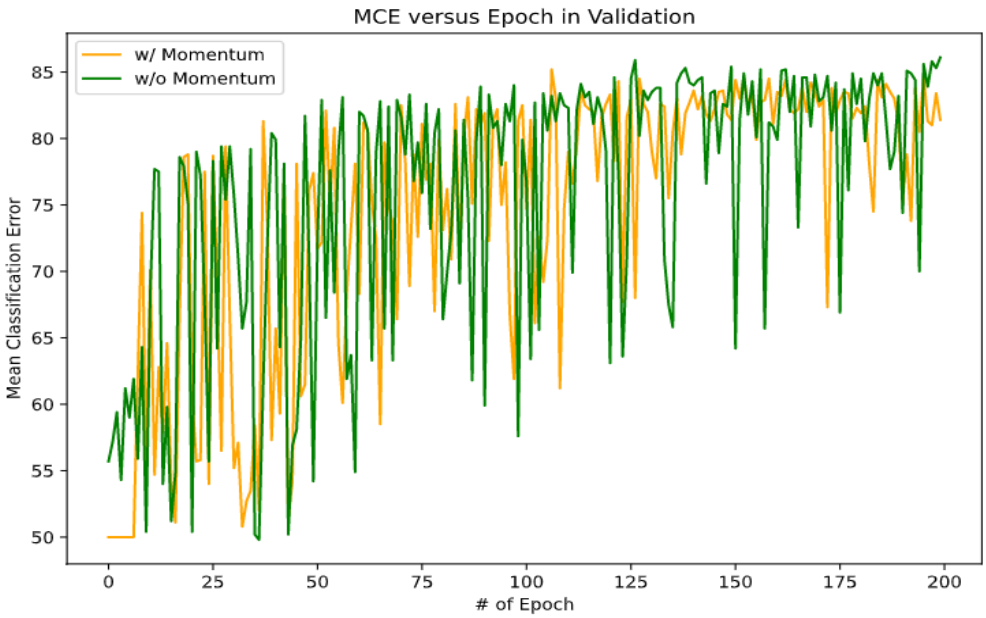
- Fast convergence
- Less oscillations



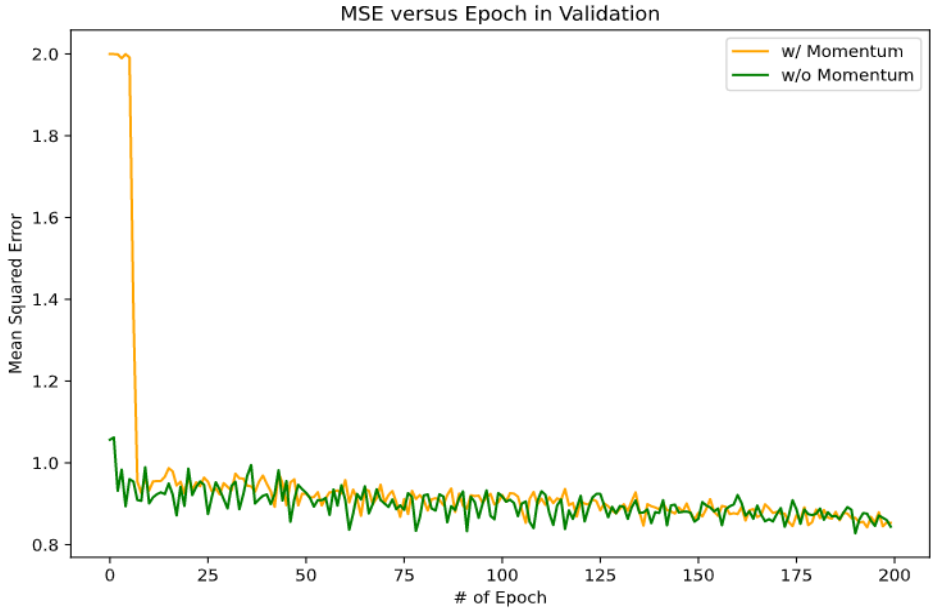
In training, we see that the model with momentum performed well and more stable w.r.t. the normal model as expected. Moreover, the momentum model convergence faster than classical one that can be seen from the figure, e.g., one can look the epoch interval 30-50, even the significantly less number of epochs, we reached 100% accuracy. Hence, momentum performed well.



In general, momentum provides fast convergence in learning process w.r.t. normal version of stochastic gradient descent. In the figure, we see that at the first epochs the model with momentum have fluctuations but then it is optimized with momentum factor and reached more stable, robust and less fluctuated results w.r.t. previous model in less number of epochs.



From the figure, we see that the model with momentum performed stable convergence with less fluctuations w.r.t. w/o momentum model as expected in validation data. Both model converged global minima but the momentum model did it with less oscillations.



As we discuss, the model with momentum learned faster even at the first epochs has little oscillations. With the increment of epoch number, the momentum model has less fluctuations w.r.t. previous one.

2 QUESTION 2

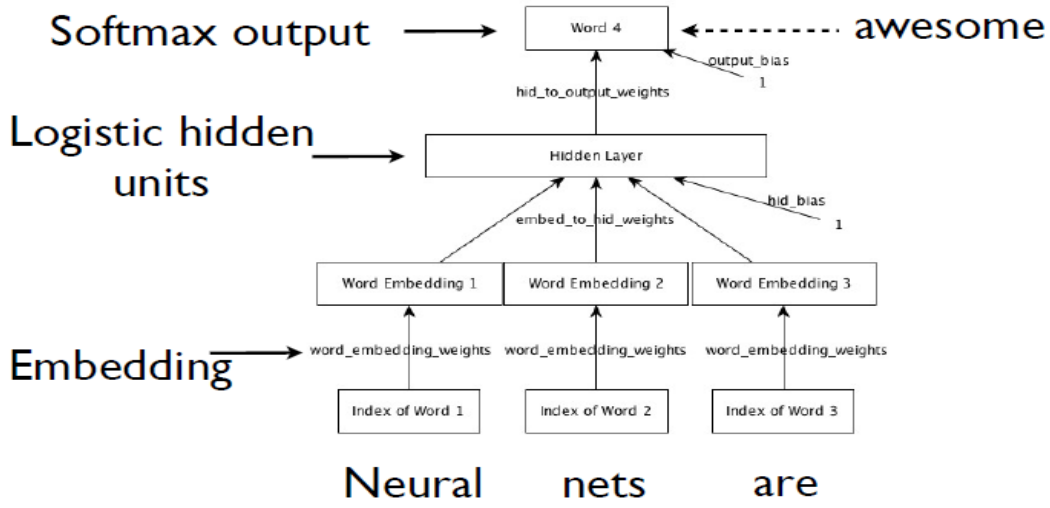
In question 2, neural networks should be trained via backpropagation to produce computational model for natural language processing. The task is to predict 4th word in sequence given preceding tri-gram, e.g., trigram: `Neural nets are', fourth word: `awesome'. The output layer predicts a separate response Z_i for each of 250 vocabulary words, and the probability of each word is estimated via a soft-max classifier.

2.1 PART A

Assuming the following parameters: a stochastic gradient descent algorithm, a mini-batch size of 200 samples, a learning rate of = 0.15, a momentum rate of = 0.85, a maximum of 50 epochs, and weights and biases initialized as random Gaussian variables of std 0.01. Lastly, the algorithm should be stopped based on the cross-entropy error on the validation data.

2.1.1 Architecture and Dataset

The following neural architecture should be implemented to train text data.



To do that, word embedding layer and hidden layer should be constructed. Embedding matrix R (250 x D) is used to linearly map each single word onto a vector representtation. In hidden units, sigmoidal activation will be used with P neurons. Since we are feeding network with 3 words, we can apply several techniques. Along this question, I am going to apply both summing and concetenating method. Concetenating input that is widely used technique along the general multi-input and single output models. Summing method will be dissucced in following parts. In the dataset, our corpus size is 250. After implementing one-hot-encode to the training and testing data, we have input matrix of 750 x 1 to feed the network. Then, to

implement feature representation embedding matrix is used to represent words in D dimension. The advantages of using word embedding layer in the neural network are reduce the dimensionalty of the input (i.e., 750 or 250 to D) and convert sparse matrix (one-hot-encode) to more computationaly eligible matrix. About the dataset, we have 372 500 training sample with their corresponding labels. To prevent computational excessive executions, I am going to apply cross-validation and early stopping based on 46 500 samples. Finally, we have 46 500 testing data to evaluate the model performance.

2.1.2 Configurable Parameters

These are configurable parameters, i.e., hyper parameters that should be tuned but question gives us constants batch size, learning rate and momentum coefficient. However, epoch number, embedding dimension, hidden layer unit should be tuned to find optimal values among given intervals.

- **Batch size** $= 200$
- **Epoch number** = max**{50}**
- **Embedding layer unit E** ϵ {32,16,8}
- **Hidden layer unit H** ϵ {256,128,64}
- **Learning rate** $\eta = 0.15$
- **Momentum Coefficient** $\alpha = 0.85$
- **Weight** $\boldsymbol{W_E, W_1, W_2}$ **and** $\boldsymbol{B_1, B_2}$ are initialized where $\boldsymbol{W_E, W_1, B_1}$ and $\boldsymbol{W_2, B_2}$ corresponds to the embedding layer, hidden layer and output layer, respectively.

$$\boldsymbol{W_E, W_1, W_2} \text{ and } \boldsymbol{B_1, B_2} \sim \mathcal{N}(\mu = 0, \sigma = 0.01)$$

The construction of the architecture and initializtion of the parameters is given below:

```
class Word2Vec:

    def __init__(self,embed_size,hidden_size):
        np.random.seed(42)
        self.lr = 0.15
        self.batch_size = 250
        self.alpha = 0.85

        self.loss_train_list = []
        self.loss_test_list = []
        self.acc_train_list = []
        self.acc_test_list = []

        self.prev_updates = {'WE':0,'W1':0,'W2':0,
                             'B1':0,'B2':0}
```

```
        self.D = embed_size
        self.P = hidden_size

        self.sample_size = data['TrainX'].shape[0]
        self.feature_size = data['TrainX'].shape[1]
        self.vocab_size = data['Corpus'].shape[0]

        self.n_update = round((self.sample_size/self.batch_size))

        self.W_emb_size = self.vocab_size,self.D
        self.W1_size = self.D,self.P
        self.W2_size = self.P,self.vocab_size
        self.B1_size = 1,self.P
        self.B2_size = 1,self.vocab_size

        self.WE =  Gauss(0,scale = 0.01, size = (self.W_emb_size))
        self.W1 = Gauss(0,scale = 0.01, size = (self.W1_size))
        self.W2 = Gauss(0,scale = 0.01, size = (self.W2_size))
        self.B1 = Gauss(0,scale = 0.01, size = (self.B1_size))
        self.B2 = Gauss(0,scale = 0.01, size = (self.B2_size))
```

2.1.3 Activation's

Along this question, sigmoidal activation is used to activate hidden layers and soft-max function is used to get probabilistic predictions from the model. Here are the mathematical expressions:

$$\text{sigmoid}(Z_i) = \frac{1}{1+ e^{-Z_i}}$$

$$\text{soft-max}(Z_i) = \frac{e^{Z_i}}{\sum_j e^{Z_j}}$$

As we see, both functions have exponential terms, but in soft-max, since the sign of the exponential argument is positive, with the increase of input magnitude, the term e^{Z_i} is going to increase exponentially that may cause problems in software implementation. (Quantization errors) Here are the Python implementations:

Sigmoid Function:

```
    def sigmoid(self,X):
        return 1/(1 + np.exp(-X))
```

Gradient of Sigmoid

```
    def sigmoid_gradient(self,X):
        return self.sigmoid(X) * (1-self.sigmoid(X))
```

Softmax Function

```
def softmax(self,X):
    e_x = np.exp(X)
    return e_x /np.sum(e_x)
```

Stable Version of Softmax

```
def softmax_stable(self,X):
    e_x = np.exp(X - np.max(X, axis=-1, keepdims=True))
    return e_x / np.sum(e_x, axis=-1, keepdims=True)
```

Numerical instability and weirdness of the soft-max function

As we can see, soft-max function includes exponential terms so with the increase of the input, soft-max function is getting overflow since NumPy package can take up maximum 10^{38}. Even though the number 10^{38} can vary machine to machine due to its capabilities, we need to make sure that soft-max output within this range. There are several ways to prevent overflows. The stable version of the soft-max function is sufficient to get results but I am going to touch upon another technique listed below:

1. **Scaling the input**

   One of the possible solution is to scale the input. It helps to soft-max to prevent from overflow just by rearranging the magnitudes of the scalars. Hence, according to my experiments, min-max scalar is doing well. Let X be the input to the soft-max, therefore, after scaling the input became:

$$X = \frac{X - \max\{X\}}{\max\{X\} - \min\{X\}}$$

   The following code shows the implementation of min-max scalar:

```
def MinMaxScaling(self,X):
    return (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
```

2. **Log-soft-max**

   Taking natural logarithm of the soft-max can helps to overflow and gives more stable results. The mathematical expression for that is:

$$\text{Log-Soft-max}(Z_i) = \log\left(\frac{e^{Z_i}}{\sum_{j=1}^{m} e^{Z_j}}\right)$$

$$= Z_i - \log\left(\sum_{j=1}^{m} e^{Z_j}\right)$$

In 2nd equation, I utilize the fact that $\log(e^{Z_i}) = Z_i$. Finally, according to my experiments, min-max scaling and log-soft-max performed well with the cross entropy loss to prevent overflow due to quantization errors. Here are log-soft-max implementations:

```
    def log_softmax_stable(self,x):
        e_x = np.exp(x - np.max(x))
        return np.log(e_x / np.sum(x,axis=0))

    def log_softmax(self,x):
        x = x - np.max(x)
        a = np.logaddexp.reduce(x)
        return np.exp(x - a)
```

2.1.4 Converting One-Hot-Encode

Since the text indexes are given in continuous form, we should convert index matrices into one-hot-encode form to feed the word2vec model. The following piece of code, transform input and labels into one-hot-encode-form:

```
class OneHotEncode:
    def __init__(self):
        self.vocab_size = 250

    def vectorize2D(self,Y):
        size = Y.shape[0]
        val = np.zeros((size,self.vocab_size))

        for i in range(size):
            val[i,int(Y[i])-1]=1
        return val

    def vectorize3D(self,X):
        size = X.shape[0]
        val1 = np.zeros((size,3,self.vocab_size))

        for i in range(size):
            for j in range(3):
                out = np.zeros(self.vocab_size)
                out[X[i,j]-1] = 1
                val1[i,j,:] = out
        return val1

    def transform(self,data,vector):
        if vector == '3D':
```

```
            return  self.vectorize3D(data)
        else:
            return  self.vectorize2D(data)
```

```
OneHotEncoder = OneHotEncode()
data['EncodedTrainY'] = OneHotEncoder.transform(data['TrainY'], vector =
'2D')
data['EncodedTestY'] = OneHotEncoder.transform(data['TestY'],vector =
'2D')
data['EncodedValY'] = OneHotEncoder.transform(data['ValY'], vector =
'2D')

data['EncodedTrainX'] = OneHotEncoder.transform(data['TrainX'], vector =
'3D')
data['EncodedTestX'] = OneHotEncoder.transform(data['TestX'],vector =
'3D')
data['EncodedValX'] = OneHotEncoder.transform(data['ValX'], vector =
'3D')
```

Final dimensions are:

```
One hot encode training data dimension     :  (372500, 3, 250)
One hot encode training label dimension    :  (372500, 250)
One hot encode validation data  dimension  :  (46500, 3, 250)
One hot encode validation label  dimension :  (46500, 250)
One hot encode testing data  dimension     :  (46500, 3, 250)
One hot encode testing label  dimension    :  (46500, 250)
```

Feeding input by summing:

```
train_data =
np.sum((data['EncodedTrainX'][:,0,:],data['EncodedTrainX'][:,1,:],data['E
ncodedTrainX'][:,2,:]),axis=0)
train_label = data['EncodedTrainY']
val_data =
np.sum((data['EncodedValX'][:,0,:],data['EncodedValX'][:,1,:],data['Encod
edValX'][:,2,:]),axis=0)
val_label = data['EncodedValY']
test_data =
np.sum((data['EncodedTestX'][:,0,:],data['EncodedTestX'][:,1,:],data['Enc
odedTestX'][:,2,:]),axis=0)
test_label = data['EncodedTestY']
```

Feeding input by concatenating:

```
train_data_conc =
np.concatenate((data['EncodedTrainX'][:,0,:],data['EncodedTrainX'][:,1,:]
,data['EncodedTrainX'][:,2,:]),axis=1)
train_label_conc = data['EncodedTrainY']
val_data_conc =
np.concatenate((data['EncodedValX'][:,0,:],data['EncodedValX'][:,1,:],dat
a['EncodedValX'][:,2,:]),axis=1)
val_label_conc = data['EncodedValY']
test_data_conc =
np.concatenate((data['EncodedTestX'][:,0,:],data['EncodedTestX'][:,1,:],d
ata['EncodedTestX'][:,2,:]),axis=1)
test_label_conc = data['EncodedTestY']
```

```
One hot encode training data dimension     :  (372500, 750)
```

Therefore, we successfully convert our input and output matrix to required form to feed the network.

2.1.5 Forward Pass

In the forward propagation, we have a 3 stage that are: word embedding layer, hidden layer and output layer. The following mathematical expressions explains the computations:

To convert word into vector representation in efficient form, $W_{embedding}$ is dot producted by input matrix X to create linear word embedding layer. In this layer, there are no activations or let's say linear activation, i.e., F(X) = X.

$$L_{embedding} = \sum_{i=1}^{N}(W_{embedding} * X)$$

$$O_{embedding} = L_{embedding}$$

Then, the output of the word embedding layer $O_{embedding}$ is dot producted by the hidden layer's weight W_1 then added by bias term B_1 to feed the activation of the hidden layer. Then, sigmoidal activation function is used to activate to the hidden layer so that output is O_{hidden}.

$$L_{hidden} = \sum_{i=1}^{N}(W_1 * O_{embedding} + B_1)$$

$$O_{hidden} = \text{sigmoid}(L_{hidden_2})$$

Finally, to get predictions from to model, we feed to the soft-max function by multiplied O_{hidden} by output layer's weights W_2 then added by bias term B_2.

$$L_{output} = \sum_{i=1}^{N}(W_2 * O_{hidden_2} + B_2)$$

$$O_{output} = \text{soft-max}(L_{output})$$

Python implementation:

```
def linear(self,inp,W):
    return np.dot(inp,W)

def forward(self,X):
    emb_linear = self.linear(X,self.WE)

    Z1 = self.linear(emb_linear,self.W1) - self.B1

    A1 = self.sigmoid(Z1)
    Z2 = self.linear(A1,self.W2) - self.B2
```

```
        A2 = self.softmax_stable(Z2)

        return {"Z1": Z1,"A1": A1,
                "Z2": Z2,"A2": A2,
```

Note that the sign of bias term does not change anything, since it will be learned anyway.

2.1.6 Loss

Then, we need to calculate a loss to evaluate the model performance. To do that, categorical cross entropy loss is used. The following mathematical equations gives the formula:

$$\text{Cross-Entropy Loss } Ł(O_{output_i}, Y_i) = \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} Y_{i_k} * \log(O_{output_{i_k}})$$

$$= \sum_{i=1}^{N} Y_i * \log(O_{output_i})$$

The following code shows implementation:

```
    def CrossEntropyLoss(self,pred,label):

        m = pred.shape[0]
        cost = -(1 / m) * np.sum(np.sum(label * np.log(pred+ 1e-3), axis=1, keepdims=True), axis=0)

        return cost
```

Note that there are several ways to implement cross-entropy wisely. The following is one example: It is also used binary cross-entropy implementations.

```
    def CrossEntropyLoss(self,pred,label):
        m = pred.shape[0]

        preds = np.clip(pred, 1e-16, 1 - 1e-16)
        loss = np.sum(-label * np.log(preds) - (1 - label) * np.log(1 -
preds))
        return loss/m
```

2.1.7 Backpropagation

In this part, I am going to gradients of the loss function w.r.t. configurable parameters. The following expression are used to calculate gradients.

Necessary Gradients

$$\frac{\partial Error}{\partial W_2} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial W_2}$$

$$\frac{\partial Error}{\partial B_2} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial B_2}$$

$$\frac{\partial Error}{\partial W_1} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden}} * \frac{\partial O_{hidden}}{\partial L_{hidden}} * \frac{\partial L_{hidden}}{\partial W_1}$$

$$\frac{\partial Error}{\partial B_2} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden}} * \frac{\partial O_{hidden}}{\partial L_{hidden}} * \frac{\partial L_{hidden}}{\partial B_2}$$

$$\frac{\partial Error}{\partial W_{Embed}} = \frac{\partial Error}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden}} * \frac{\partial O_{hidden}}{\partial L_{hidden}} * \frac{\partial L_{hidden}}{\partial O_{Embed}} * \frac{\partial O_{Embed}}{\partial L_{Embed}} * \frac{\partial L_{L_{Embed}}}{\partial W_{Embed}}$$

Computed Gradients:

$$\frac{\partial Error}{\partial W_2} = (O_{output} - Y) * (O_{hidden}{}^T)$$

$$\frac{\partial Error}{\partial B_2} = \sum_{i=1}^{N} ((O_{output} - Y))$$

$$\frac{\partial Error}{\partial W_1} = (O_{output} - Y)\,(W_3{}^T) * (sigmoid(L_{hidden})) * O_{embed}{}^T$$

$$\frac{\partial Error}{\partial B_2} = \sum_{i=1}^{N} ((O_{output} - Y)\,(W_3{}^T) * (sigmoid(L_{hidden}))$$

$$\frac{\partial Error}{\partial W_{Embed}} = (O_{output} - Y) * (W_3{}^T) * \big(sigmoid(L_{hidden})\big) * (W_2{}^T) * (X^T)$$

Therefore, we can give gradients to the optimizer of the model that is stochastic gradient descent.

```
    def linear_gradients(self,inp,delta):
        return { 'dW' : self.linear(inp,delta)/self.batch_size,
                 'dB' : np.sum(delta, axis=0,
keepdims=True)/self.batch_size}



    def cross_entropy_gradient(self, preds, label):
        preds = np.clip(preds, 1e-15, 1 - 1e-15)
        grad_ce = - (label/preds) + (1 - label) / (1 - preds)
        return grad_ce

    def softmax_stable_gradient(self, X):
        soft_out = self.softmax_stable(X)
        return soft_out * (1 - soft_out)

    def backward(self,outs, X, Y):

        E  = outs['E']
        Z1 = outs['Z1']
        A1 = outs['A1']
        Z2 = outs['Z2']
        A2 = outs['A2']
```

```
        dZ2 = self.cross_entropy_gradient(A2,Y) *
self.softmax_stable_gradient(Z2)
        dW2 = self.linear_gradients(A1.T,dZ2)['dW']
        dB2 = self.linear_gradients(A1.T,dZ2)['dB']

        dZ1 = self.linear(dZ2,self.W2.T) * self.sigmoid_gradient(Z1)
        dW1 = self.linear_gradients(E.T ,dZ1)['dW']
        dB1 = self.linear_gradients(E.T ,dZ1)['dB']

        dEmbed = self.linear(dZ1,self.W1.T)
        dWE = self.linear_gradients(X.T,dEmbed)['dW']

        return {"dW1": dW1, "dW2": dW2,
                "dB1": dB1, "dB2": dB2,
                "dWE": dWE}
```

Note that when the cross-entropy loss is used by soft-max functions there is a shortcut of doing this.

$$\frac{\partial CrossLoss}{\partial L_{output}} = \frac{\partial CrossLoss}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} = (O_{output} - \mathrm{Y})$$

2.1.8 Stochastic Gradient Descent with momentum

To optimize the model, stochastic gradient descent on mini-batches with the momentum is used. Here are the update terms and update process.

W_2 update

$$\Delta {W_2}^N = -\eta * \frac{\partial Error}{\partial W_2} + \alpha * \Delta {W_2}^{N-1}$$

$$W_2 \mathrel{+}= \Delta {W_2}^N$$

W_1 update

$$\Delta {W_1}^N = -\eta * \frac{\partial Error}{\partial W_1} + \alpha * \Delta {W_1}^{N-1}$$

$$W_1 \mathrel{+}= \Delta {W_1}^N$$

B_2 update

$$\Delta {B_2}^N = -\eta * \frac{\partial Error}{\partial B_2} + \alpha * \Delta {B_2}^{N-1}$$

$$B_2 \mathrel{+}= \Delta {B_2}^N$$

B_1 update

$$\Delta {B_1}^N = -\eta * \frac{\partial Error}{\partial B_1} + \alpha * \Delta {B_1}^{N-1}$$

$$B_1 \mathrel{+}= = \Delta {B_1}^N$$

W_{embed} update

$$\Delta {W_{Embed}}^N = -\eta * \frac{\partial Error}{\partial W_{Embed}} + \alpha * \Delta {W_{Embed}}^{N-1}$$

$$W_{Embed} \mathrel{+}= \Delta {W_{Embed}}^N$$

The following code explains the code implementation of SGD with momentum.

```
    def SGD(self,grads):
        delta_E  = -self.lr * grads['dWE'] +  self.alpha *
self.prev_updates['WE']
        delta_W1 = -self.lr * grads['dW1'] +  self.alpha *
self.prev_updates['W1']
        delta_W2 = -self.lr * grads['dW2'] +  self.alpha *
self.prev_updates['W2']
        delta_B1 = -self.lr * grads['dB1'] +  self.alpha *
self.prev_updates['B1']
        delta_B2 = -self.lr * grads['dB2'] +  self.alpha *
self.prev_updates['B2']

        self.WE += delta_E
        self.W1 += delta_W1
        self.W2 += delta_W2
        self.B1 += delta_B1
        self.B2 += delta_B2


        self.prev_updates['WE'] = delta_E
        self.prev_updates['W1'] = delta_W1
        self.prev_updates['W2'] = delta_W2
        self.prev_updates['B1'] = delta_B1
        self.prev_updates['B2'] = delta_B2

        pass
```

2.1.9 Training and Cross Validation

I implemented training loop and cross validation to decide whether stop early or not to prevent overfitting the model. To do that, I use the approach of thresholding, i.e., I decided to set the threshold value ş that means that, if the distance between training cross-entropy and validation cross-entropy is more than ş, I will stop the training since we are starting to over fit. Here is the Python code for both training, cross-validation and early stopping:

```
    def fit(self,X,Y,X_val,y_val,epochs,verbose=True, crossVal = False):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """


        for epoch in range(epochs):
            perm = np.random.permutation(self.sample_size)
            print(f'Epoch : {epoch + 1}')

            for i in range(self.n_update):

                batch_start  =  i * self.batch_size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]
                X_feed = X[index]
                y_feed = Y[index]
```

```
                outs_train = self.forward(X_feed)
                grads = self.backward(outs train,X feed,y feed)
                self.SGD(grads)
```

This is general training loop and the early stopping is implemented below:

```
                if crossVal:
                    stop = self.cross_validation(X,val_X,Y,val_Y,
threshold = 5)                 if stop:
                        break
```

The logic behind this is to stop the training if boolean stop value gets true. The true condition is specified below:

```
    def cross_validation(self,train_data,val_data,label_train,label_val,
threshold):
        train_preds = self.predict(train_data)
        val_preds = self.predict(val_data)
        train_loss = self.CrossEntropyLoss(train_preds,label_train)
        val_loss = self.CrossEntropyLoss(val_preds,label_val)

        if train_loss - val_loss < threshold:
            return True
        return False
```

Then, here are accuracy and cross-entropy is keep to analyze history of the training after.

```
cross_loss_train = self.CrossEntropyLoss(outs_train['A2'],y_feed)
predictions_train = self.predict(X)
acc_train = self.accuracy_score(predictions_train,np.argmax(Y,1))
cross_loss_val = self.CrossEntropyLoss(X_val,y_val)
predictions_val = self.predict(X_val)
acc_val = self.accuracy_score(predictions_val,np.argmax(y_val,1))
```

where predict method is given by:

```
    def predict(self,X):
        feed = self.forward(X)
        return np.argmax(feed['A2'],axis=1)
```

and accuracy score is given by:

```
    def accuracy_score(self,preds,label):
        expand = 100
        count = 0
        size = label.shape[0]
        for i in range(size):
            if preds[i] == label[i]:
                count +=1
        return  expand * (count/size)
```

2.1.10 Hyperparamaters Tuning

In this part, I tuned the configurable parameters mostly manually since training of the model takes time. Here are the code and final results:

The little grid search is implemented to tune hyperparameters.

```
best_model = None
best_val = -1
lr = [0.01,0.05,0.1,0.15,0.3,0.5,0.1,0.17]
batch_size = [50,100,150,200,250,150,250,100]
alpha = [0.1,0.3,.5,0.7,0.85,0.9,0.6,0.4]
acc = []
for i in range(8):
    model = Word2Vec(embed_size=32,hidden_size=256,lr=lr[i],
batch_size=batch_size[i],alpha=alpha[i])
        model.fit(X_feed,data['TrainY'],30)
        preds = model.predict(X_test); labels =
np.argmax(data['TestY'],1)
        val_acc = accuracy(preds,labels)
        acc.append(val_acc)

        if best_val < val_acc :
            best_val = val_acc
            best_model = model
```

Hyperparamaters	Results
Embedding size	32
Hidden layer size	256
Batch size	200
Learning rate	0.15
Momentum Coefficient	0.85
Mean of weights	0
Standard deviation of weights	1e-2
Epoch number	50

2.1.11 Results/Discuss

This question was an unstable, tricky and very challenging question for me since I debugged the code for days and tried every possible technique to get better results that trigger me to learn lots of things. Here are my trials:

Recall that we have a corpus size of 250, embedding size of D, and hidden layer size of P and given that D,P ∈ {(32,256),(16,128),(8,64)}. I experiment with these tuple pairs and my network works fine in D = 32, P =256 which is discussed above. The tuple fair D, P (8,64) is

not sufficient for the input matrix to make feature representation since we have embedding size of 8. Then the network is not converged well. For the pair of (16,128) is also works well for my model. It is also admissible the work on and one advantage of the pair is computationally less costly. But, the most optimized model is performed on the pair (32,256). So I go on with these pair.

For weights and bias initialization, I tried zero, uniform, Gauss, Xavier and He initialization to find optimal values. Best results are given when the weights are initialized by Xavier technique. Note that I am not used Xavier in the code since assignment specified the distribution. Then, I tried stochastic gradient descent w/o momentum, Adam, RMSprop, AdaGrad optimizers to optimize the hyperparamaters. The results are nearly same; network did not react well. In the Adam optimizer, the results are slightly better than RMSprop and SGD with momentum. AdaGrad performed worse among optimizers. Since these are not our topic for this paper, I am not going to include analytics and code. After that, I tried scaling and normalization technique to prevent vanishing gradient problem in sigmoidal activation and overflow in soft-max classifier. As I discussed, even quantization error caused by overflow depends on the hardware specifics of the machines, NumPy has a maximum limit of 10^{38} floating numbers. To do that, I tried min-max scaler, data standardization then I implement my own batch normalization layer in both feed forwarding and back warding to test my result. It works to prevent my model to overflow, since I was getting nan values from soft-max function and also it works for the sigmoid activation. Since before the sigmoidal activation we have 2 dot products, input can be large with time so normalizing input worked. I tried logarithmic soft-max to stabilize my network. The mathematical proof this done in above. After that, I tried different learning rates and momentum coefficients, embedding size, hidden size to experiment with the results. To do that, I implement my own random search, grid search. To go further, I tried scikit-learn hyperparamaters tuning package gridsearchCV to see results and compare between mine. Then, I tried Ray Tune that is industry level tuning technology to investigate my finding and compare optimization results between mine and Ray Tune. Furthermore, I tried transfer learning based natural language processing method by using pre-trained nlp model that I found from the GitHub then trained again over our training data to see where I may improve it. After that, I used regularization technique L_1, L_2 and dropout to experiment. None of which works well, but dropout performed slightly better on validation data as expected. But the model does not react as expected and performed nearly same that was a frustration for me. Finally, I tried different inputting techniques such as

summing and concatenating. Summing the one-dot-encoded sparse matrix column wise is a worse technique to feed the input to the word2vec network since information loss is inevitable, i.e., the sequence of the words is loss, e.g., neural networks are being the same as are network neural or network are neural even the questions accepts. Concatenating is a widely used technique to feed network multi-inputs, to do that, I concatenate my one-hot-encoded input matrix column wise, i.e., now we have 750 column vector per input. Therefore, I tried lots of things to optimize and stabilize the model, some of which are worked, other don't. The reason behind why I go deeper in optimization is to get more accuracy on given data. Here are final results and discuss:

Embedding Size D, hidden size P such that (D, P) = (32,256)

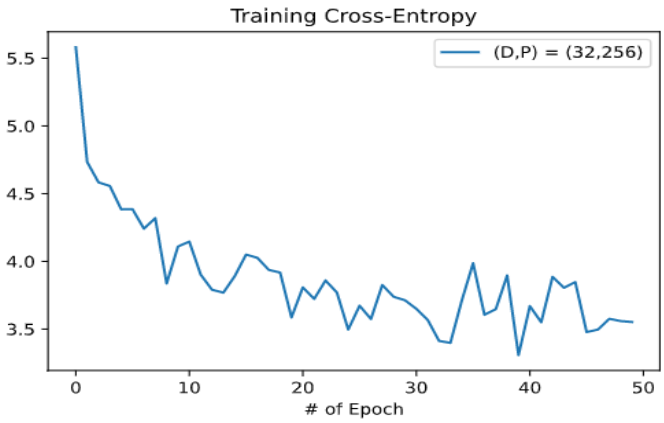
I begin with (D, P) = (32,256) and it was the best network among others here are running and results:

```
model = Word2Vec(32,256)
model.fit(train_data,train_label,test_data,test_label,50)


model_history = model.history()
```

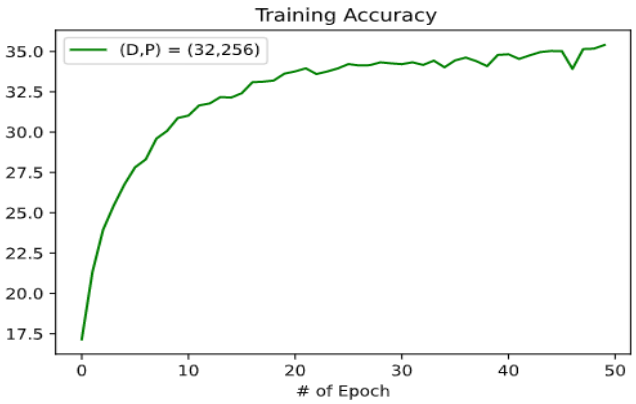
Here the evolving process of the model:

```
Epoch : 0
[0/50] ------> Training :  Accuracy: 17.152751677852347
[0/50] ------> Testing  :  Accuracy: 16.946236559139784
Epoch : 1
[1/50] ------> Training :  Accuracy: 21.321610738255035
[1/50] ------> Testing  :  Accuracy: 21.066666666666666
Epoch : 2
[2/50] ------> Training :  Accuracy: 23.946577181208053
[2/50] ------> Testing  :  Accuracy: 23.587096774193547
Epoch : 3
[3/50] ------> Training :  Accuracy: 25.449395973154367
[3/50] ------> Testing  :  Accuracy: 25.33978494623656
Epoch : 4
[4/50] ------> Training :  Accuracy: 26.754093959731545
[4/50] ------> Testing  :  Accuracy: 26.608602150537635
Epoch : 5
[5/50] ------> Training :  Accuracy: 27.819060402684563
[5/50] ------> Testing  :  Accuracy: 27.9505376344086
Epoch : 6
[6/50] ------> Training :  Accuracy: 28.31114093959732
[6/50] ------> Testing  :  Accuracy: 28.43010752688172
Epoch : 7
[7/50] ------> Training :  Accuracy: 29.598120805369128
[7/50] ------> Testing  :  Accuracy: 29.391397849462365
```

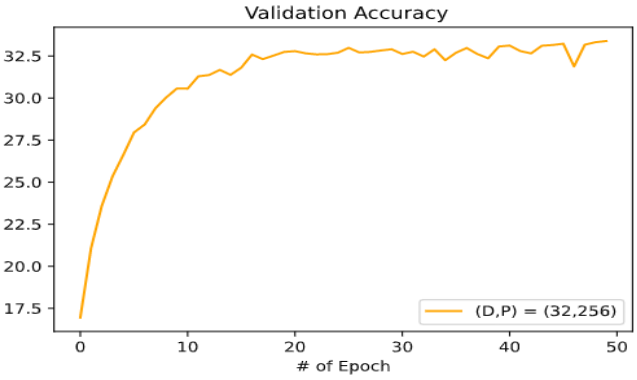
We see that our training and validation data is converging similarly that is a good sign so that the model can be generalize later.



I trained my model over 50 epochs, and we see that our loss function cross-entropy is decreasing, although there are oscillations.



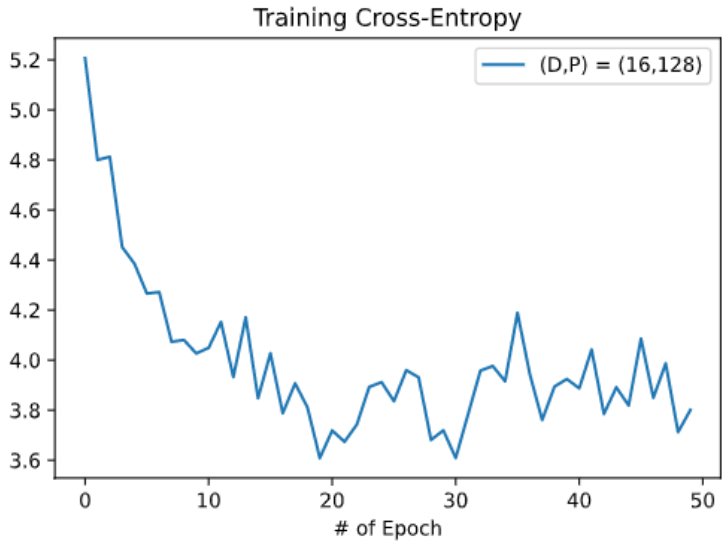
Then, training accuracy reached 37% accuracy in 50 epochs.



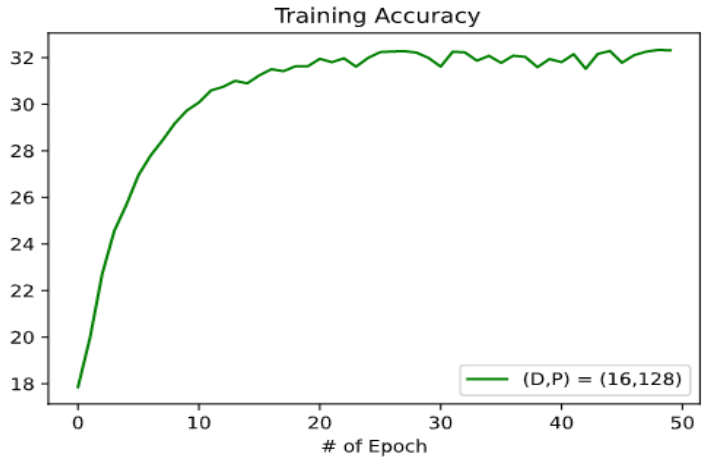
While training, I cross-validate my model to ensure that it is not over fitted. Here, in the figure above, we see that our validation accuracy is ~34% accuracy. Hence, training and validation accuracies are pretty similar.

Embedding Size D, hidden size P such that (D, P) = (16,128)

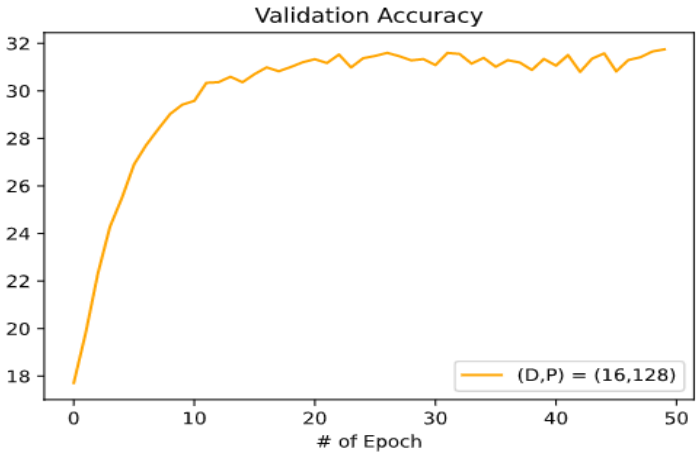
Then, I tried (D, P) = (16,128) to experiment with. Here are results and discuss:



As we see from the figure that the model with embedding size 16, hidden size 128 is performed sufficient but more oscillatory. In most of the epochs, the cross-entropy loss is decreasing anyway.



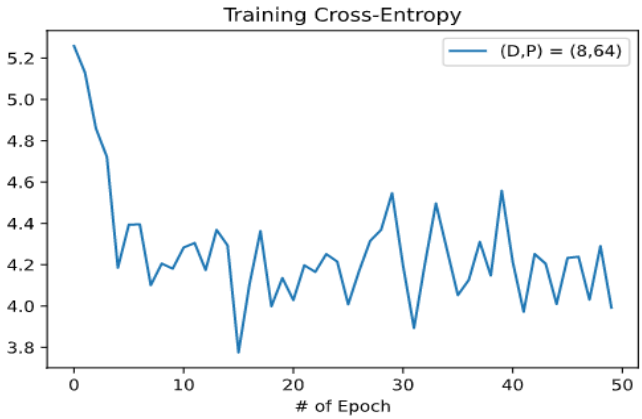
Then, training accuracy reached 32%, it is slightly less than previous model.



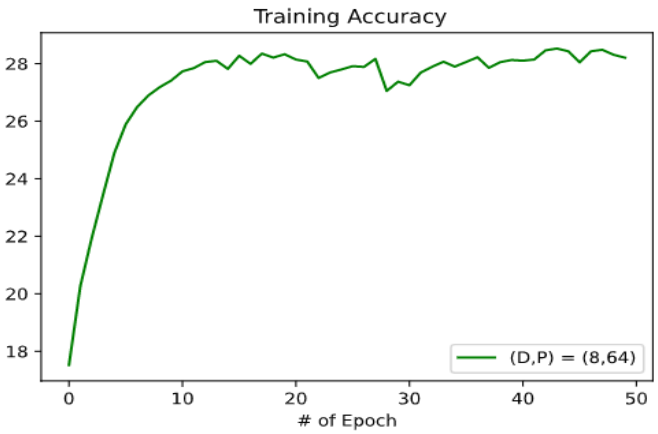
The good news is this model also reached 33% accuracy so that in the testing set this model also performed well. Hence, training and testing accuracies are very close so that we are not over fitted.

Embedding Size D, hidden size P such that (D, P) = (8,64)

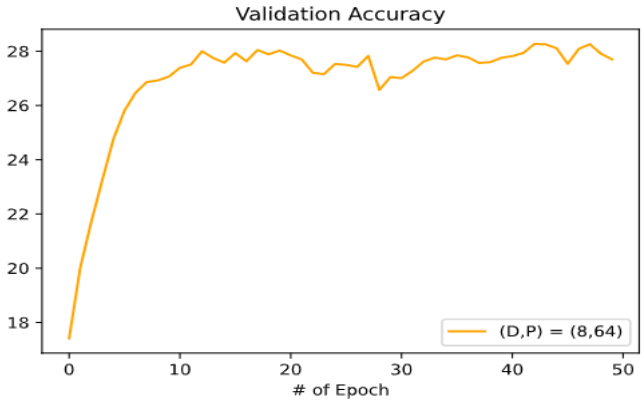
Finally, I experiment with the model (D, P) = (8,64). This was the worse model among other the reason is discussed below.



As we see from the figure, cross-entropy is not stable, even it is decreasing in the first 15 epochs. We can inference that training and validation accuracy will be less.



As expected, we see that our training accuracy reached 28% so that it is worse.



We performed similar performance in the validation data and converged 28% accuracy also. Even the model is performed worse, not over fitted.

2.2 PART B

In this part, prediction should be done via inputting trigrams and returning 4th word given sequence. Every prediction's likelihood should be stored and we should list top 10 candidates for the 4th word. To do that, I randomly select 5 sample in testing set then, I take the highest 10 probabilities with correspond their corresponding index and feed them to words data to corresponding string. Here are results and code:

In the testing set, I reached the ~ 33% accuracy.

```
# Note that np.random.seed() is not placed to see different results in
differents runnings
test_preds = model.predict(test_data)
test_acc = model.accuracy_score(test_preds,np.argmax(test_label,1))
print(f'Test Accuracy: {test_acc}')
```

```
33.38924731182796
```

Then, to predict probability matrix, I feed the input to the feed forward function.

```
forward = model.forward(test_data)
probs_softmax = forward['A2']
```

Then, from 46500 samples, I random select 5 sample.

```
num_sample = test_data.shape[0]
n_predict = 5
random_idx = np.random.randint(num_sample,size = n_predict)
```

Then, I sliced the 5 random samples from both prediction probability matrix and the corresponding words in the test data.

```
# 5 random sample's probabilities
five_random_num_probs = probs_softmax[random_idx]
random_five_words =
data['Corpus'][data['TestX'][random_idx]].astype('U13')
```

Then, I find the indexes maximum probability of 10 value in randomly selected 5 sample.

```
def top_k_preds(probs,k,n_predict):
    top_k = np.zeros((n_predict,k))
    for i in range(n_predict):
         top_k[i] = probs.argsort()[i][-k:][::-1]
    return top_k
```

After that, I find out the corresponding strings in the dictionary.

```
top_10_preds = top_k_preds(five_random_num_probs,10,5)
top_10_words = data['Corpus'][top_10_preds.astype('int')].astype('U13')
```

The rest is printing top 10 predicted candidates for given words:

```
_____________________________________________________________________________

Input word 1 -----> day, in, only
Top 10 predicted candicates : go, be, ., do, get, see, come, know, think, work
_____________________________________________________________________________

Input word 2 -----> an, might, left
Top 10 predicted candicates : other, money, time, only, last, right, best, place, way, game
_____________________________________________________________________________

Input word 3 -----> many, then, only
Top 10 predicted candicates : do, go, get, be, think, take, play, come, know, make
_____________________________________________________________________________

Input word 4 -----> american, not, still
Top 10 predicted candicates : ., back, there, to, down, out, home, want, from, in
_____________________________________________________________________________

Input word 5 -----> including, university, well
Top 10 predicted candicates : nt, ., he, not, she, i, and, it, too, do
_____________________________________________________________________________
```

Actually, the network predictions are sensible and it was a surprising for me. Especially, in 2^{nd} and 3^{rd} trigram, the pairs are pretty good. One can construct meaningful sentences from these words easily.

3 QUESTION 3

The goal of this question is to introduce you fully-connected neural networks, and various optimization and regularization choices for these models by experimenting with two Python demos, one on a fully-connected network model, and a second on dropout regularization on such a network.

3.1 PART A

In this part, I am going to explain each of parts of fully-connected neural nets ipynb files by clipping the picture of notebook then commenting. The reason behind this question is to be familiarized with the fully-connected neural nets by designing them modular so that one can construct arbitrary neural architecture.

Fully-Connected Neural Nets

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

In this exercise we will implement fully-connected networks using a more modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

This part initializes the concept to begin.

```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

The necessary packages are imported to work on.

```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

The data used for this assignment is well-known CIFAR10 dataset that have 10 classes with 4900 training sample with 32 x 32 pixel in 3 color channel. Moreover, we have 1000 validation and testing data to evaluate the performance of the future models.

In this part, dense layer is implemented via weighted sum of the inputs with weights and then added by bias term.

Affine layer: foward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769847728806635e-10
```

Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b, dout)

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

When training neural networks, numerical evaluation of gradients w.r.t. some parameters are necessary, so in this part necessary numerical gradients are found.

ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

# Compare your output with ours. The error should be on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

Rectified linear unit is used for activation of the layers and ReLu is widely used one since its simplicity, performance and gradient calculations.

ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be on the order of e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

Since ReLU used for activating the layers, the gradients should be calculated to backward propagating.

Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour?

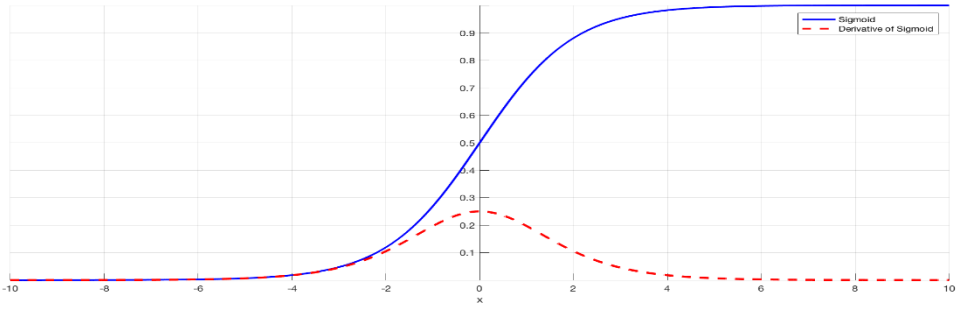
1. Sigmoid
2. ReLU
3. Leaky ReLU

3.1.1 Inline Question 1 Answer

In this part, I am going to discuss each of the activation functions differently. Before that, let me initialize the concept. Vanishing gradients is a problem in the context of back propagation in neural network. Vanishing gradients is simply that gradients are getting or approaching zero that cause problem of preventing the weights from changing its value so that network cannot learn anymore. As we know, gradient descent should be performed while training to update to the weights with the gradients coming from back propagation, but if the update is zero or close to 0, this means there are no changes in the configurable parameters of the network so the network is stopped learning.

3.1.1.1 Sigmoid Activation

In sigmoidal activation, the function squishes a large input into 0 to 1 hence large in magnitude inputs in both negative and positive sign can cause to the problem of vanishing gradients. From the figure below, we can see that as we close to y = 0 and y = 1, there are little changes in the outputs so since the function is nearly constant, corresponding partial derivative is becoming zero.

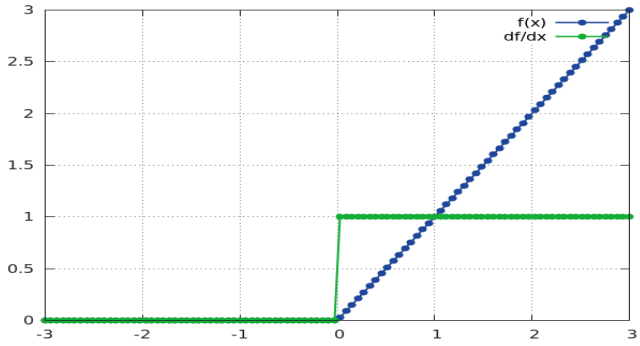


In simple feed forward neural network, this may not be a big problem but in more complicated architectures, it may cause serious problems since back propagation is found by continuously calculating its partial derivatives and then applying chain rule to find results, the gradients decrease exponentially as we down to the initial layers that will come up with vanishing gradients. One dimensional example can be positive x value that corresponding y value close to 1 and negative x values that corresponding y value is close to 0. Some simple implementations like batch normalization, min-max scaling get helps functions to prevents vanishing gradients.

3.1.1.2 Rectified Linear Unit (ReLU)

ReLU is a widely used activation function since its capabilities as we discuss earlier. ReLU has not a vanishing gradient problem when x>0 and its one of the main reason why ReLU is

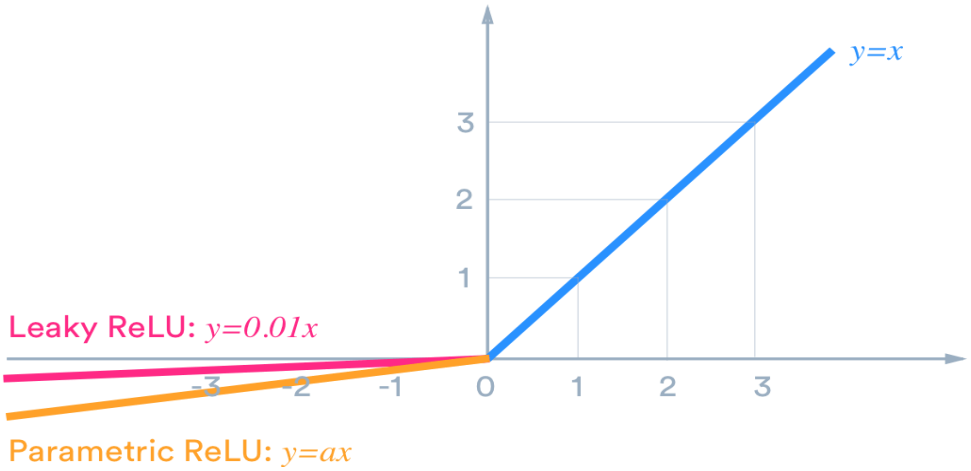
used heavily in deep learning context. Firstly, ReLU is not continuously differentiable. At x=0, the breaking point between x and 0, the gradient cannot be computed. This is not too problematic, but can very lightly impact training performance. From the figure, we see that blue points represent the ReLU function, it is 0 until 0, then follows y = x line that can be expressed as ReLU = max {0, L} where L is the linear input of $Wx + b$. So, one of the main advantages is the reduced the likelihood of vanishing gradients after 0.



The other benefit of ReLUs is sparsity. Sparsity arises when before 0. The more such units that exist in a layer the sparser the resulting representation. Sigmoidal activations on the other hand are always likely to generate some non-zero value resulting in dense representations. Sparse representations seem to be more beneficial than dense representations. Moreover, ReLU is computationally efficient function since its simplicity. Finally, ReLu activations tend to show better performance in learning processes of the models, i.e., better converging the global minima. One cons can be said about the ReLU is dying ReLu problem, i.e., if there are too many activations that is below 0, ReLU outputs zero so that training of the model is facing problems. Therefore, ReLU can suffer from vanishing gradient from when x<0, and have a perfectly defined gradient when x>0. One dimensional example can be any value in x<0, e.g., -1, 2,….

3.1.1.3 Leaky ReLU

Leaky ReLU is another version of ReLU activations with the progresses on the problem of dying ReLU. As we discuss, ReLU may be problematic in some cases that have lots of zeros of the outputs of the layer in the network. Leaky ReLU comes into play with corrected version of ReLU where slop is changed with left side of the 0. Leaky ReLU also does not have a vanishing gradient problem since its derivative is well-defined and non-zero almost everywhere (except close to 0).



The formula of leaky ReLu is the following:

$$F(x) = \{0.01x, \text{ if } x<0$$

$$\{x, \text{ else}$$

Therefore, leaky ReLU is another activation function that has no problem with vanishing gradient most of the cases (except x = 0). One dimensional example can be 0.

I am going to continue to explain the notebook below.

"Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0], b, dout)

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  6.750562121603446e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

Sandwich layers are generally used for combination of linear activation following by nonlinearity activation. In terms of code efficiency, they can be more practical since they are generally used together, i.e., linear activation and nonlinear activation).

To reconstruct modular multi-layer perceptron (MLP) model, previously implemented parts are reminded.

Two-layer network

In the previous assignment you implemented a two-layer neural network in a single monolithic class. Now that you have implemented modular versions of the necessary layers, you will reimplement the two layer network using these modular implementations.

Open the file `cs231n/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'
```

This part is previously implemented and included for continuity.

```
model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.52e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 8.18e-07
W2 relative error: 2.85e-08
b1 relative error: 1.09e-09
b2 relative error: 7.76e-10
```

As we said, to implement more modular neural architecture, we constructed solver class to train data.

Solver

In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

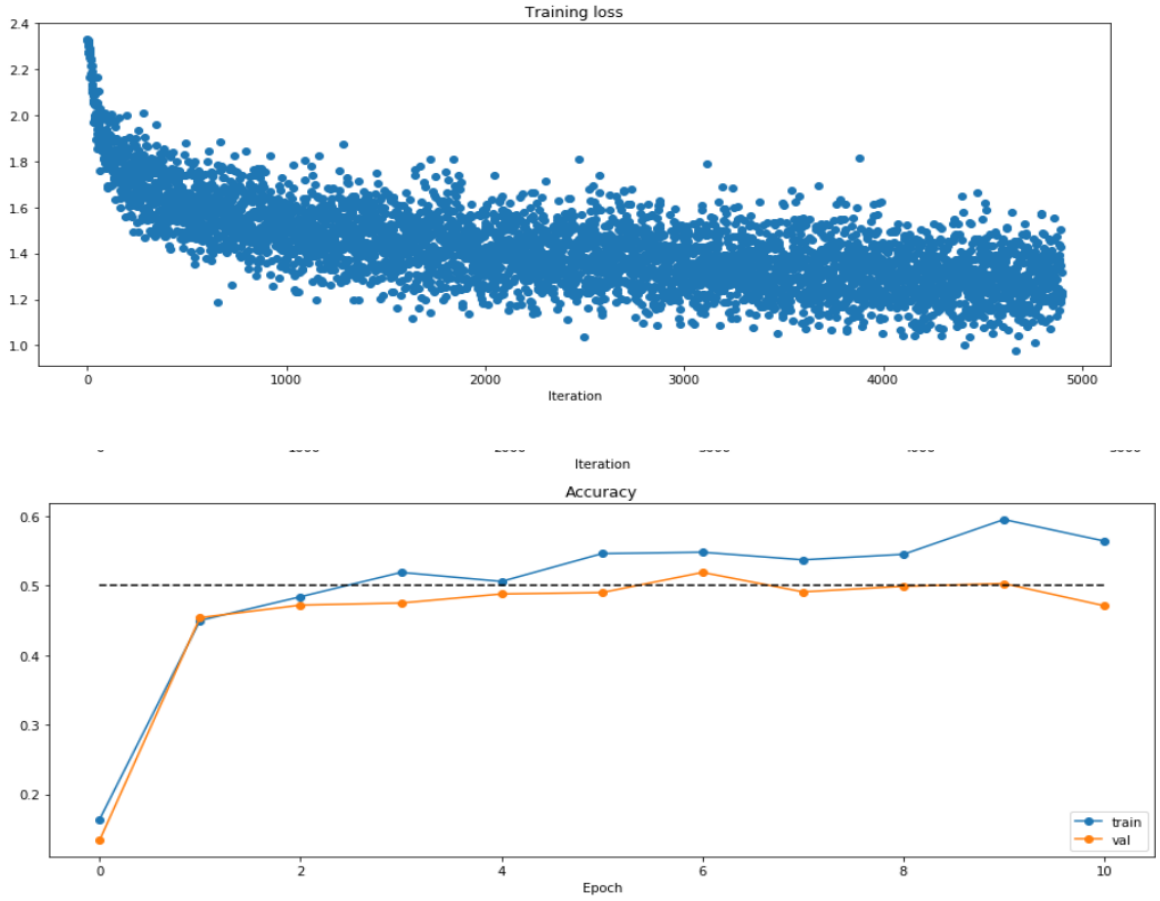
Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

```
model = TwoLayerNet()
solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                        #
##############################################################################

model = TwoLayerNet(hidden_dim=100, reg=0.2)
solver = Solver(model, data, update_rule='sgd',
                optim_config={'learning_rate': 1e-3}, lr_decay=0.95,
                num_epochs=10, batch_size=100, print_every=100)
solver.train()

##############################################################################
#                              END OF YOUR CODE                              #
##############################################################################
```

```
(Iteration 1 / 4900) loss: 2.332096
(Epoch 0 / 10) train acc: 0.164000; val_acc: 0.134000
(Iteration 101 / 4900) loss: 1.857220
(Iteration 201 / 4900) loss: 2.000576
```

Here are the results of the two-layer neural network.

```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



The loss and accuracy as expected. The model performed on approximately 60% on training data and 50% on validation data. It seems model get over fitted a bit.

Multilayer network

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cs231n/classifiers/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout or batch/layer normalization; we will add those features soon.

Initial loss and gradient check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. Do the initial losses seem reasonable?

For gradient checking, you should expect to see errors around 1e-7 or less.

```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64)

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

  # Most of the errors should be on the order of e-7 or smaller.
  # NOTE: It is fine however to see an error for W2 on the order of e-5
  # for the check when reg = 0.0
```

```
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

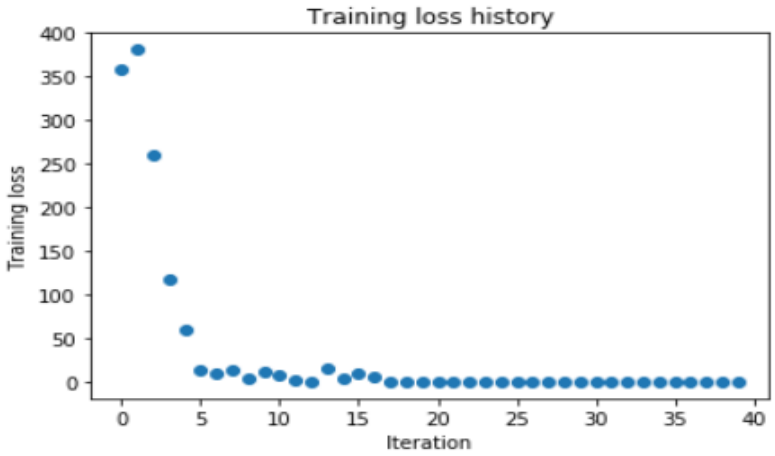
```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 7.36e-09
W2 relative error: 6.87e-08
W3 relative error: 3.48e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.80e-10
```

This is one of the previous implemented snippet of code to test to code.

As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the learning rate and initialization scale to overfit and achieve 100% training accuracy within 20 epochs.

```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

weight_scale = 1e-1
learning_rate = 1e-3
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

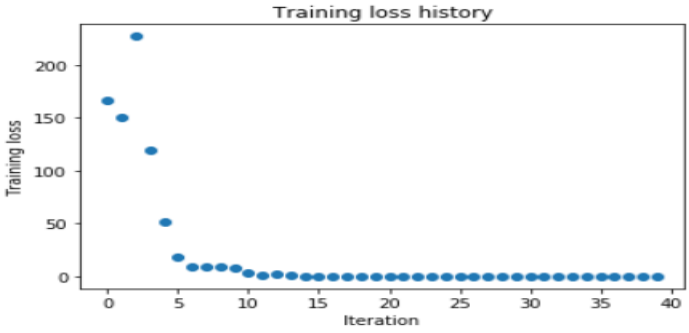
```
(Iteration 1 / 40) loss: 357.428290
(Epoch 0 / 20) train acc: 0.220000; val_acc: 0.111000
(Epoch 1 / 20) train acc: 0.380000; val_acc: 0.141000
(Epoch 2 / 20) train acc: 0.520000; val_acc: 0.138000
(Epoch 3 / 20) train acc: 0.740000; val_acc: 0.130000
(Epoch 4 / 20) train acc: 0.820000; val_acc: 0.153000
(Epoch 5 / 20) train acc: 0.860000; val_acc: 0.175000
(Iteration 11 / 40) loss: 6.726589
```



As the notebooks say, accuracy became 100% in 20 epochs.

```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

learning_rate = 2e-3
weight_scale = 1e-1
model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 166.501707
(Epoch 0 / 20) train acc: 0.100000; val_acc: 0.107000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.101000
(Epoch 2 / 20) train acc: 0.160000; val_acc: 0.122000
(Epoch 3 / 20) train acc: 0.380000; val_acc: 0.106000
(Epoch 4 / 20) train acc: 0.520000; val_acc: 0.111000
(Epoch 5 / 20) train acc: 0.760000; val_acc: 0.113000
(Iteration 11 / 40) loss: 3.343141
```



The reason to train mini data with 5-layer network is to see the model's overfitting. We get 100% accuracy in training data but we have 12% accuracy in validation.

3.1.2 Inline Question 2 Answer

Inline Question 2:

Did you notice anything about the comparative difficulty of training the three-layer net vs training the five layer net? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

Since the number of configurable parameters are increased with the number of hidden layer, the model with more hidden layer requires more computational power. After that, the models with more hidden layers can be comparatively difficult to control since we cannot have a deep inside about the hidden layer activations. After that, the concept of sensitivity Analysis in neural networks is an approach of understanding of initialization of configurable parameters

between the effects of the output. Initializing weights and biases in correct form, i.e., correct variance and mean, affects the mean size of the activation in following layers. When comparing three-layer network with five layer, assuming the other necessary parameters are similar, the initialization scale can be more sensitive in the model with higher number of layer, i.e., the effects of the initialization has more effect on the learning process of the model. In model with higher layers, since there are comparatively more configurable parameters, each parameter affected exponentially of initialization scale of the parameters. Scaling the network parameters, i.e., its mean and variance, can have deeper effect on the complex network w.r.t. shallow networks. Finding correct mean and variance can be found by search algorithm such as random search, manual search and grid search.

From that point, we will move into optimizer in the context of deep learning. There are some widely used optimization algorithms such as AdaGrad, RMSprop and Adam. They all adaptive optimization algorithms that means they adapted the process of learning by rearranging learning rate so that we can reach more efficiently and faster to the global minima. Here are examples and implementations.

Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent. See the Momentum Update section at http://cs231n.github.io/neural-networks-3/#sgd for more information.

Open the file `cs231n/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

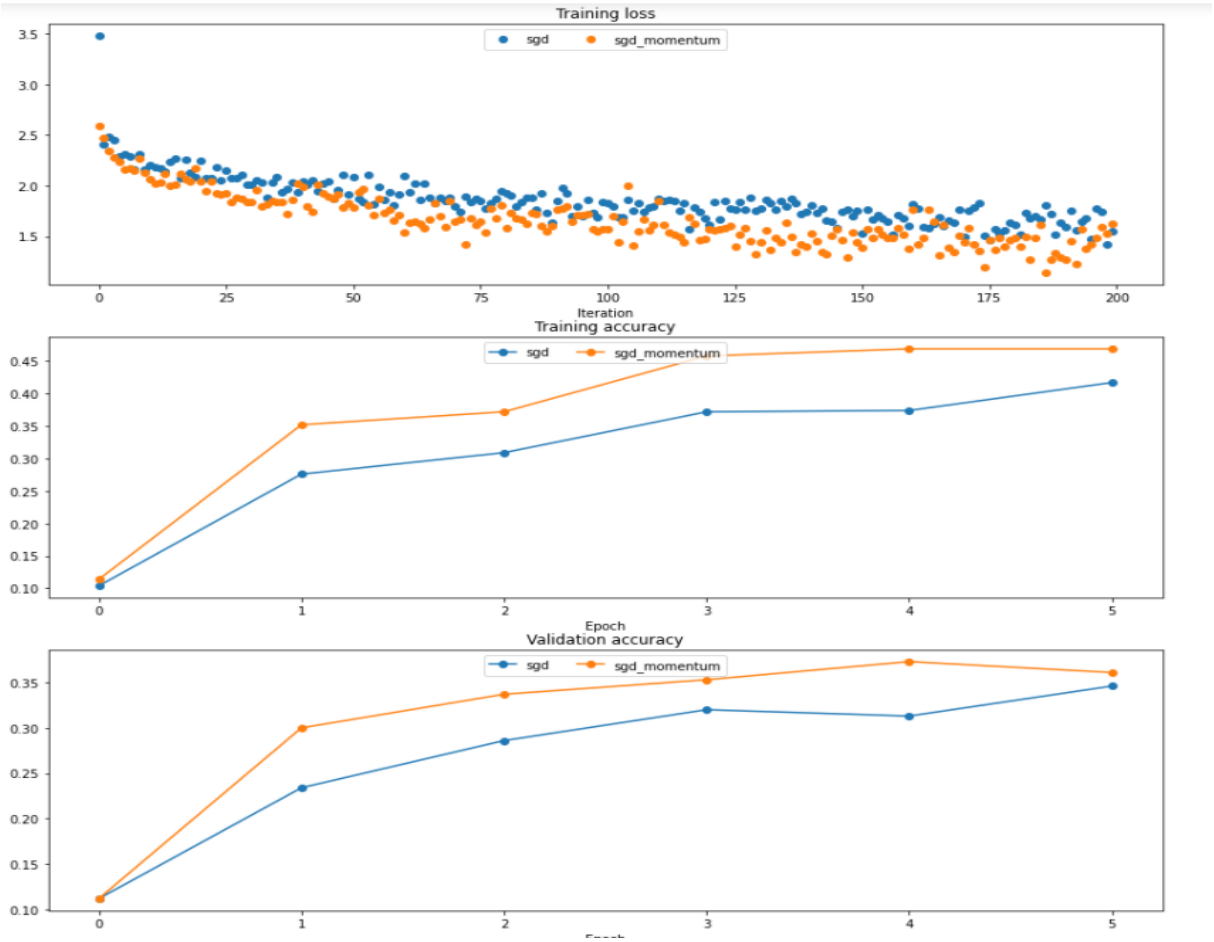
```
from cs231n.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

# Should see relative errors around e-8 or less
print('next_w error: ', rel_error(next_w, expected_next_w))
print('velocity error: ', rel_error(expected_velocity, config['velocity']))
```

```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

Stochastic gradient descent with momentum is a optimization algorithms in neural network that I implement in both question 1 and 2.

Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}

for update_rule in ['sgd', 'sgd_momentum']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': 1e-2,
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)
```



As expected, momentum learning is more effective than vanilla gradients descent since update history. In below, there are more complex and effective optimization algorithms are provided.

RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `cs231n/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

NOTE: Please implement the *complete* Adam update rule (with the bias correction mechanism), not the first simplified version mentioned in the course notes.

[1] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning 4 (2012).

[2] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

```
: # Test RMSProp implementation
  from cs231n.optim import rmsprop

  N, D = 4, 5
  w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
  dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
  cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

  config = {'learning_rate': 1e-2, 'cache': cache}
  next_w, _ = rmsprop(w, dw, config=config)

  expected_next_w = np.asarray([
    [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
    [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
    [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
    [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]])
  expected_cache = np.asarray([
    [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
    [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
    [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
    [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]])

  # You should see relative errors around e-7 or less
  print('next_w error: ', rel_error(expected_next_w, next_w))
  print('cache error: ', rel_error(expected_cache, config['cache']))
```

```
next_w error:  9.524687511038133e-08
cache error:  2.6477955807156126e-09
```

```
# Test Adam implementation
from cs231n.optim import adam

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('v error: ', rel_error(expected_v, config['v']))
print('m error: ', rel_error(expected_m, config['m']))
```

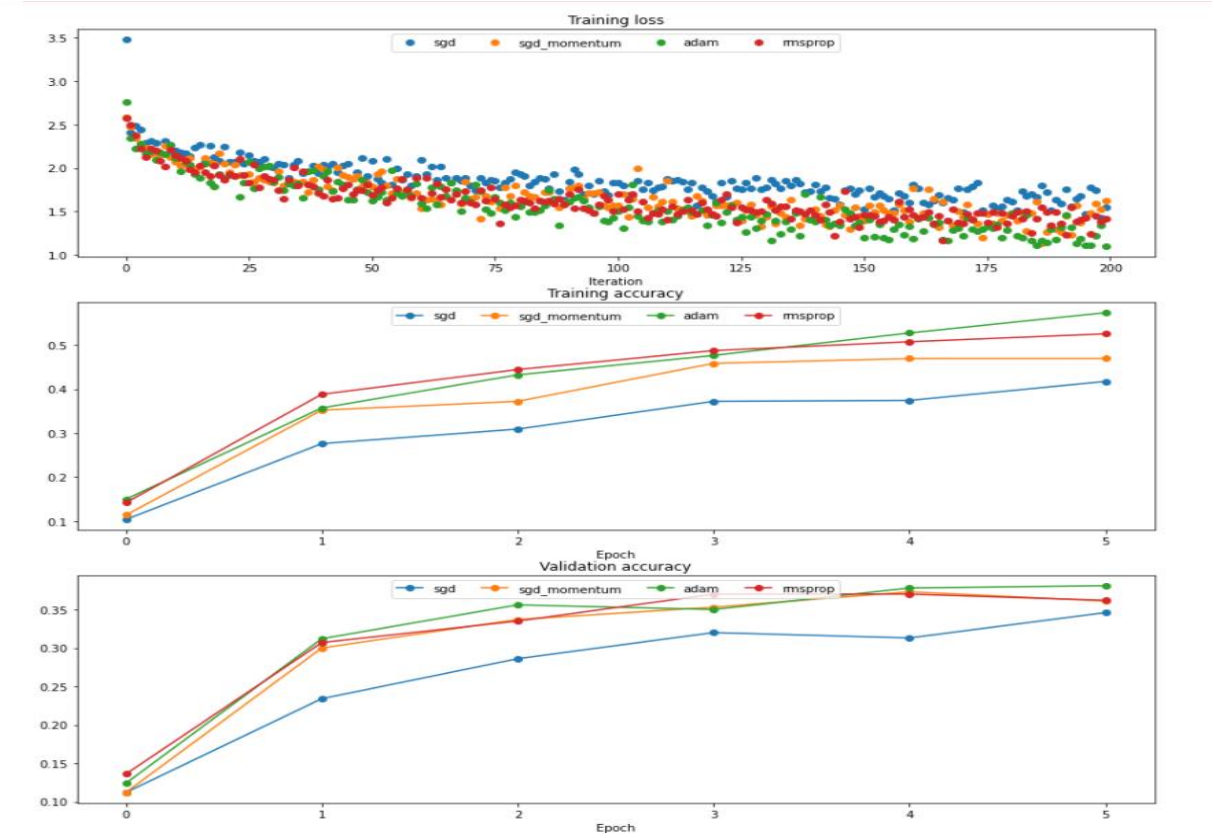
```
next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

RMSprop and Adam optimizers are widely used algorithms to reach the optimal learning curves and both are adaptive learning algorithms since the adapt to learning process by configuring update parameter.

```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': learning_rates[update_rule]
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  adam
(Iteration 1 / 200) loss: 3.476928
(Epoch 0 / 5) train acc: 0.126000; val_acc: 0.110000
(Iteration 11 / 200) loss: 2.027712
(Iteration 21 / 200) loss: 2.183358
```



These are results of the several optimization algorithms; we see that Adam optimizer performed well among other as expected since its more efficient algorithm. The reason behind is explained below in terms of mathematical perspective. Secondly, RMSprop performed also well on this data, it is nearly give the same result with the Adam optimizer. Moreover, we inference from the figures that adding momentum term accelerates and optimize the learning curve better than vanilla stochastic gradient descent. The mathematical explanations are provided below.

Inline Question 3:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

3.1.3 Inline Question 3 Answer

AdaGrad optimization algorithm keeps track of the sum of the squared gradients hat decay the learning rate for parameters in propagation to their update history. The mathematical expression for that:

$$\delta_i = \delta_{i-1} + \quad \nabla^2\theta_i$$

$$\theta_i := \theta_{i-1} - \frac{\eta}{\sqrt{\delta_i}+\varepsilon} * \nabla\theta_i$$

Where δ_i is the cumulative sum of squared gradients, $\nabla\theta_i$ is the gradient of the θ_i that is configurable parameters of the network and ε is the scalar to prevent zero division. When the cumulative sum increases gradually, the denominator of the update term increases simultaneously that cause to decreases the overall update term $\frac{\eta}{\sqrt{\delta_i}+\varepsilon} * \nabla\theta_i$. With the decreases of the update term, AdaGrad optimization is becoming slower. Therefore, AdaGrad stuck when close to convergence since cumulative sum is increases gradually so that overall update term is significantly decreases.

RMSprop optimization algorithms fix the issue of AdaGrad by multiplying decaying rate to the cumulative sum and enables to forget the history of cumulative sum after certain point that depends on the decaying term that helps to convergence to global minima. The mathematical expression for RMSprop is given by:

$$\delta_i = \alpha * \ \delta_{i-1} + \quad (1-\alpha) * \nabla^2\theta_i$$

$$\theta_i := \theta_{i-1} - \frac{\eta}{\sqrt{\delta_i + \varepsilon}} * \nabla\theta_i$$

where α is the decaying terms. Therefore, this decaying term provides faster convergence and forget to cumulative sum of gradient history that results in more optimized solution.

Adam: optimization algorithm is the developed version of the RMSprop by taking first and second momentum of the gradient separately. Therefore, Adam also fixes the slow convergence issue in close the global minima. In this version of adaptive algorithm, the mathematical expression is given by:

$$\delta_{M_i} = \beta_1 * \delta_{M_i} + (1 - \beta_1) * \nabla\theta_i$$

$$\delta_{V_i} = \beta_2 * \delta_{V_i} + (1 - \beta_2) * \nabla^2\theta_i$$

$$\widetilde{\delta_{M_i}} = \frac{\delta_{M_i}}{1-\beta_1}$$

$$\widetilde{\delta_{V_i}} = \frac{\delta_{V_i}}{1-\beta_2}$$

$$\theta_i := \theta_{i-1} - \frac{\eta}{\sqrt{\widetilde{\delta_{V_i}} + \varepsilon}} * \widetilde{\delta_{M_i}}$$

where δ_{M_i} is the first moment decaying cumulative sum of gradients, δ_{V_i} is the second moment decaying cumulative sum of gradients, $\widetilde{\delta_{M_i}}$ and $\widetilde{\delta_{V_i}}$ are bias corrected values of δ_{M_i} and δ_{V_i} and ŋ is the learning rate. Finally, we are training of the fully connected model with CIFAR-10 dataset.

Train a good model!

Train the best fully-connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully-connected net.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the assignment we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional nets rather than fully-connected nets.

You might find it useful to complete the `BatchNormalization.ipynb` and `Dropout.ipynb` notebooks before completing this part, since those techniques can help you train powerful models.

```
best_model = None
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################

hidden_dims = [100] * 4

range_weight_scale = [1e-2, 2e-2, 5e-3]
range_lr = [1e-5, 5e-4, 1e-5]

best_val_acc = -1
best_weight_scale = 0
best_lr = 0

print("Training...")

for weight_scale in range_weight_scale:
    for lr in range_lr:
        model = FullyConnectedNet(hidden_dims=hidden_dims, reg=0.0,
                                  weight_scale=weight_scale)
        solver = Solver(model, data, update_rule='adam',
                        optim_config={'learning_rate': lr},
                        batch_size=100, num_epochs=5,
                        verbose=False)
        solver.train()
        val_acc = solver.best_val_acc

        print('Weight_scale: %f, lr: %f, val_acc: %f' % (weight_scale, lr, val_acc))

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            best_weight_scale = weight_scale
            best_lr = lr
            best_model = model
```

```
Training...
Weight_scale: 0.010000, lr: 0.000010, val_acc: 0.341000
Weight_scale: 0.010000, lr: 0.000500, val_acc: 0.501000
Weight_scale: 0.010000, lr: 0.000010, val_acc: 0.327000
Weight_scale: 0.020000, lr: 0.000010, val_acc: 0.407000
Weight_scale: 0.020000, lr: 0.000500, val_acc: 0.515000
Weight_scale: 0.020000, lr: 0.000010, val_acc: 0.401000
Weight_scale: 0.005000, lr: 0.000010, val_acc: 0.277000
Weight_scale: 0.005000, lr: 0.000500, val_acc: 0.492000
Weight_scale: 0.005000, lr: 0.000010, val_acc: 0.225000
Best val_acc: 0.515000
Best weight_scale: 0.020000
Best lr: 0.000500
```

Test your model!

Run your best model on the validation and test sets. You should achieve above 50% accuracy on the validation set.

```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Validation set accuracy:  0.515
Test set accuracy:  0.525
```

Therefore, with the Adam optimizer, we reach the 52.5% accuracy on the test set and 51.5% accuracy in the validation set. We can see the effects of Adam optimizer w.r.t. stochastic gradient descent that will implemented in previous assignment on the CIFAR-10 dataset.

3.2 PART B

In this part of the assignment, I am going to dive into dropout regularization in neural networks. Dropout is regularization technique that based on the random selection of neurons in the layers then activate or de-activate with probability p. This probability determines the how many neurons will be activated in training. For example, let p = 0.5, let H be the neuron unit in layer L, therefore in every iteration $\frac{H}{2}$ number of neuron is selected randomly, and activated while other $\frac{H}{2}$ is de-activated. Let's start to analyze the given notebook.

Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some features to zero during the forward pass. In this exercise you will implement a dropout layer and modify your fully-connected network to optionally use dropout.

[1] Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012

```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
run the following from the cs231n directory and try again:
python setup.py build_ext --inplace
You may also need to restart your iPython kernel
```

As we did before, let's look at the shape of the training data CIFAR-10.

```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

```
X_train:  (49000, 3, 32, 32)
y_train:  (49000,)
X_val:  (1000, 3, 32, 32)
y_val:  (1000,)
X_test:  (1000, 3, 32, 32)
y_test:  (1000,)
```

As I say, we have 49 000 training images with 32x32 pixels in 3 color channels RGB. Moreover, we have 1000 validation and 1000 testing images to evaluate the model's performance.

Since dropout can be regarded an another layer in the network, we can create dropout layer for both forward and backpropagation.

Dropout forward pass

In the file `cs231n/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

```
np.random.seed(231)
x = np.random.randn(500, 500) + 10

for p in [0.25, 0.4, 0.7]:
  out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
  out_test, _ = dropout_forward(x, {'mode': 'test', 'p': p})

  print('Running tests with p = ', p)
  print('Mean of input: ', x.mean())
  print('Mean of train-time output: ', out.mean())
  print('Mean of test-time output: ', out_test.mean())
  print('Fraction of train-time output set to zero: ', (out == 0).mean())
  print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
  print()
```

```
Running tests with p =  0.25
Mean of input:  10.000207878477502
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.749784
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  10.000207878477502
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.600796
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  10.000207878477502
Mean of train-time output:  9.987811912159426
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.30074
Fraction of test-time output set to zero:  0.0
```

The Python implementation of dropout layer is providing below. Note that we need to consider dropout layer for training and testing separately. Since in testing we don't want to drop any hidden unit, we will consider separately.

```
def dropout_forward(x, dropout_param):
    """
    Performs the forward pass for (inverted) dropout.

    Inputs:
    - x: Input data, of any shape
    - dropout_param: A dictionary with the following keys:
      - p: Dropout parameter. We keep each neuron output with probability p.
      - mode: 'test' or 'train'. If the mode is train, then perform dropout;
        if the mode is test, then just return the input.
      - seed: Seed for the random number generator. Passing seed makes this
        function deterministic, which is needed for gradient checking but not
        in real networks.

    Outputs:
    - out: Array of the same shape as x.
    - cache: tuple (dropout_param, mask). In training mode, mask is the dropout
      mask that was used to multiply the input; in test mode, mask is None.

    NOTE: Please implement **inverted** dropout, not the vanilla version of dropout.
    See http://cs231n.github.io/neural-networks-2/#reg for more details.

    NOTE 2: Keep in mind that p is the probability of **keep** a neuron
    output; this might be contrary to some sources, where it is referred to
    as the probability of dropping a neuron output.
    """
    p, mode = dropout_param['p'], dropout_param['mode']
    if 'seed' in dropout_param:
        np.random.seed(dropout_param['seed'])

    mask = None
    out = None

    if mode == 'train':


        mask = (np.random.rand(*x.shape) < p) / p
        out = x * mask


    elif mode == 'test':


        out = x


    cache = (dropout_param, mask)
    out = out.astype(x.dtype, copy=False)
```

In this code, we just filter the out array with probability p, means give 0 for de-activated and 1 for activated. Then, simple multiply with the out with mask to filter it.

Then, we need to implement a backward propagation to find gradients as always.

Dropout backward pass

In the file `cs231n/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

```
np.random.seed(231)
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx, dropout_param)[0], x, dout)

# Error should be around e-10 or less
print('dx relative error: ', rel_error(dx, dx_num))
```

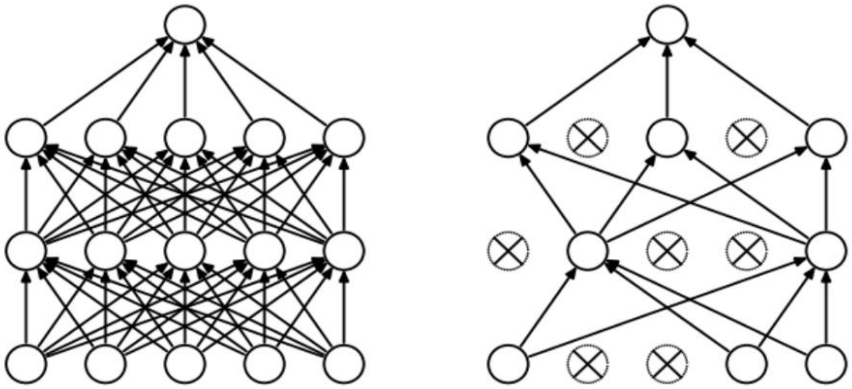
```
dx relative error:  5.44560814873e-11
```

Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

3.2.1 Inline Question 1 Answer

When our model is in the training H * p number of neuron is activated that result in scaled by p. After training, we are in testing mode, we don't want to any dropout layer and the weights corresponds to the layer is different in different trainings. To overcome this issue, in inverse dropout case, we scale the weights by p in forward propagation to resolve any problem that can be faced in testing phase. In a nutshell, if we don't divide term by p, we could not take average of weights to be used in testing phase, we just got the summation of the weights that come from the different training phase of the model result in large in scale weights. The representative scheme is given below:



(a) Standard Neural Net (b) After applying dropout.

After that, let's dive into action.

Fully-connected nets with Dropout

In the file `cs231n/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the net receives a value that is not 1 for the `dropout` parameter, then the net should add dropout immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for dropout in [1, 0.75, 0.5]:
  print('Running check with dropout = ', dropout)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            weight_scale=5e-2, dtype=np.float64,
                            dropout=dropout, seed=123)

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

  # Relative errors should be around e-6 or less; Note that it's fine
  # if for dropout=1 you have W2 error be on the order of e-5.
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  print()
```

```
Running check with dropout =  1
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11

Running check with dropout =  0.75
Initial loss:  2.302371489704412
W1 relative error: 1.90e-07
W2 relative error: 4.76e-06
W3 relative error: 2.60e-08
b1 relative error: 4.73e-09
b2 relative error: 1.82e-09
b3 relative error: 1.70e-10
```

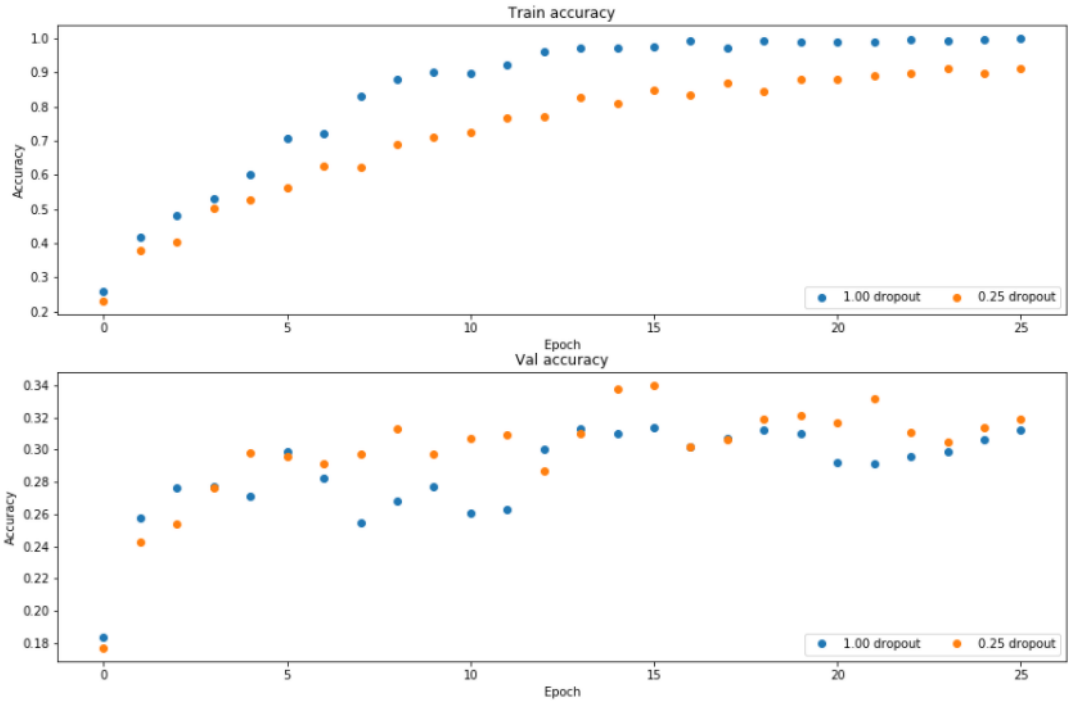
Regularization experiment

As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training and validation accuracies of the two networks over time.

```
# Train two identical nets, one with dropout and one without
np.random.seed(231)
num_train = 500
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}
dropout_choices = [1, 0.25]
for dropout in dropout_choices:
  model = FullyConnectedNet([500], dropout=dropout)
  print(dropout)

  solver = Solver(model, small_data,
                  num_epochs=25, batch_size=100,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
```

```
1
(Iteration 1 / 125) loss: 7.856644
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.276000
(Epoch 3 / 25) train acc: 0.532000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.600000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.708000; val_acc: 0.299000
(Epoch 6 / 25) train acc: 0.722000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.255000
(Epoch 8 / 25) train acc: 0.880000; val_acc: 0.268000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.277000
```

```
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for dropout in dropout_choices:
  solver = solvers[dropout]
  train_accs.append(solver.train_acc_history[-1])
  val_accs.append(solver.val_acc_history[-1])

plt.subplot(3, 1, 1)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].train_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.gcf().set_size_inches(15, 15)
plt.show()
```



The results of the network are given above. We see that with the dropout layer with probability 0.25, we get better results in generalization phase so that we avoid overfitting comparatively. Even the training accuracy of model with dropout is not much w.r.t. fully-connected model, we get better performance on testing so that the gap between training and testing accuracy is decrease that results in preventing model from overfitting.

Inline Question 2:

Compare the validation and training accuracies with and without dropout -- what do your results suggest about dropout as a regularizer?

Answer:

Inline Question 3:

Suppose we are training a deep fully-connected network for image classification, with dropout after hidden layers (parameterized by keep probability p). How should we modify p, if at all, if we decide to decrease the size of the hidden layers (that is, the number of nodes in each layer)?

Answer:

3.2.2 Inline Question 2 Answer

Dropout is regularization technique to prevent overfitting the model. Basically, this technique prevents hidden neurons from co-adapting too much so that every neuron in this layer work similarly. We can see that the expected results from the figure below. In the case of training, the training accuracy of the model without dropout is higher but in the validation case, reached less accuracy that cause model to over fit. So the model with dropout with p = 0.25, performed better in validation set, so we basically avoid overfitting. Hence, our generalization error is improved.

3.2.3 Inline Question 3 Answer

In the case of decrement of the hidden units in hidden layer, there are two approaches:

- Keep constant the probability p

  When reducing the hidden units, keeping constant probability p works since p is independent of the hidden layers size, i.e., let L_{hidden} be the hidden layer and the dimension $L_{hidden} \in \mathbb{R}^{256}$ so that we have 256 neurons in that layer. Then, let p = 0.5 that represents the keep 50% of the values in training. Therefore, in each iteration we have 128 randomly selected neurons. Then, let decrease the size of the $L_{hidden} \in \mathbb{R}^{128}$ with the same probability p hence we have 64 active neurons in each iterations. Therefore, dropout rate is arranging the hidden size proportionally. Since we want to get smaller network, this approach is sensible.
- Gradually increase the probability p

  When reducing the hidden units, we may want to increase p to keep same amount of hidden neurons activated. For example, L_{hidden} be the hidden layer and the dimension $L_{hidden} \in \mathbb{R}^{256}$ so that we have 256 neurons in that layer, then let's say the size of the layer is decreased to $L_{hidden} \in \mathbb{R}^{192}$ .Initially, we have dropout rate 0.5 so that 128 neurons is activated, then let's increase the p to $\frac{2}{3}$ ($\frac{2}{3}$ of the neurons active) so that we

have a 128 active neurons in each iteration. Since we want to simplify the model to prevent overfitting, this may not be very helpful since we keeping same neurons in that layer and the neurons tend to co-adapt to network more that we don't want.

4 APPENDIX

4.1 Q1 CODE

```
# To add a new cell, type '# %%'
# To add a new markdown cell, type '# %% [markdown]'
# %%
from IPython import get_ipython

# %%
import numpy as np
from numpy.random import normal as Gauss
from numpy.random import randn
import matplotlib.pyplot as plt
import h5py
get_ipython().run_line_magic('matplotlib', 'inline')


# %%
class DatasetLoader():

    def __init__(self, path):
        self.path = path

    def load(self):
        X_train,y_train,X_test,y_test = self.get_data(self.path)
        X_train = self.normalize(self.flatten_images(X_train))
        X_test = self.normalize(self.flatten_images(X_test))

        return X_train,y_train,X_test,y_test

    def flatten_images(self,X):
        x = X.reshape(X.shape[0],-1)
        return x

    def normalize(self,X):
        return X/255

    def get_info(self):
        print(f"Training images has a shape  : {X_train.shape}  and
contains {X_train.shape[0]} training images with 32x32 pixel" )
        print(f"Training labels has a shape  : {y_train.shape}")
        print(f"Testing images  has a shape  : {X_test.shape}   and
contains {X_test.shape[0]} testing images with 32x32 pixel" )
        print(f"Testing labels  has a shape  : {y_test.shape}")



    def get_data(self,path) -> tuple :
        """
        Given the path of the dataset, return
        training and testing images with respective
```

```
        labels.
        """

        with h5py.File(path,'r') as F:
            # Names variable contains the names of training and testing
file
            names = list(F.keys())

            X_train = np.array(F[names[2]][:])
            y_train = np.array(F[names[3]][:])
            X_test = np.array(F[names[0]][:])
            y_test = np.array(F[names[1]][:])

            y_train = y_train.reshape(y_train.shape[0],1)
            y_test = y_test.reshape(y_test.shape[0],1)

            y_train[y_train==0]=-1
            y_test[y_test==0]=-1

        return X_train,y_train,X_test,y_test

path = 'assign2_data1.h5'
data = DatasetLoader(path)
X_train,y_train,X_test,y_test = data.load()


# %%
data.get_info()


# %%
class TwoLayerNetwork:

    def __init__(self,input_size = X_train.shape,batch_size = 19 ,n_neurons
= 76 , mean = 0,std = 1, lr = 1e-1, distribution = 'Xavier'):
        np.random.seed(15)
        self.lr = lr
        self.mse_train = {}
        self.mce_train = {}
        self.mse_test = {}
        self.mce_test = {}

        self.sample_size = input_size[0]
        self.feature_size = input_size[1]
        self.batch_size = batch_size
        self.n_neurons = n_neurons
        self.mean,self.std = mean,std

        self.dist = distribution


        self.n_update = round((self.sample_size/self.batch_size))

        self.W1_size = self.feature_size,self.n_neurons
        self.W2_size = self.n_neurons,1

        self.B1_size = 1,self.n_neurons
        self.B2 size = 1,1

        self.B1 = Gauss(loc = self.mean, scale = self.std, size =
(self.B1_size)) * 0.01
```

```
        self.B2 = Gauss(loc = self.mean, scale = self.std, size =
(self.B2_size)) * 0.01


        self.he_scale1 = np.sqrt(2/self.feature_size)
        self.he_scale2 = np.sqrt(2/self.n_neurons)
        self.xavier_scale1 = np.sqrt(2/(self.feature_size+self.n_neurons))
        self.xavier_scale2 = np.sqrt(2/(self.n_neurons+1))



        if (self.dist == 'Zero') :
            self.W1 = np.zeros((self.W1_size))
            self.W2 = np.zeros((self.W2_size))

        elif (self.dist == 'Gauss'):
            self.W1 = Gauss(loc = self.mean, scale = self.std, size =
(self.W1_size))* 0.01
            self.W2 = Gauss(loc = self.mean, scale = self.std, size =
(self.W2_size))* 0.01

        elif (self.dist == 'He'):
            self.W1 = randn(self.W1_size[0],self.W1_size[1]) *
self.he_scale1
            self.W2 = randn(self.W2_size[0],self.W2_size[1]) *
self.he_scale2

        elif (self.dist == 'Xavier'):

            self.W1 = randn(self.W1_size[0],self.W1_size[1]) *
self.xavier_scale1
            self.W2 = randn(self.W2_size[0],self.W2_size[1]) *
self.xavier_scale2



    def forward(self,X):

        Z1 = (X @ self.W1)  + self.B1
        A1 = np.tanh(Z1)
        Z2 = (A1 @ self.W2) + self.B2
        A2 = np.tanh(Z2)

        return {"Z1": Z1,"A1": A1,"Z2": Z2,"A2": A2}


    def tanh(self,X):
        return (np.exp(X) - np.exp(-X))/(np.exp(X) + np.exp(-X))

    def tanh_der(self,X):
        return  1-(np.tanh(X)**2)

    def backward(self,outs, X, Y):
        m = (self.batch_size)

        Z1 = outs['Z1']
        A1 = outs['A1']
        Z2 = outs['Z2']
        A2 = outs['A2']

        dZ2 = (A2-Y)* self.tanh_der(Z2)
```

```
        dW2 = (1/m) * (A1.T @ dZ2)
        dB2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)

        dZ1 = (dZ2 @ self.W2.T) * self.tanh_der(Z1)
        dW1 = (1/m) * (X.T @ dZ1)
        dB1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)


        return {"dW1": dW1, "dW2": dW2,
                "dB1": dB1, "dB2": dB2}

    def Loss(self,pred,y_true):

        mse = np.square(pred-y_true).mean()

        pred[pred>=0]=1
        pred[pred<0]=-1

        mce = (pred == y_true).mean()

        return {'MSE':mse,'MCE':mce}


    def SGD(self,grads):
        self.W1 -= self.lr * grads['dW1']
        self.W2 -= self.lr * grads['dW2']
        self.B1 -= self.lr * grads['dB1']
        self.B2 -= self.lr * grads['dB2']

    def fit(self,X,Y,X_test,y_test,epochs = 300,verbose=True):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """

        m = self.batch_size

        for epoch in range(epochs):
            perm = np.random.permutation(self.sample_size)

            for i in range(self.n_update):


                batch_start = i * m
                batch_finish = (i+1) * m
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]


                outs = self.forward(X_feed)
                loss = self.Loss(outs['A2'],y_feed)

                outs_test = self.forward(X_test)
                loss_test = self.Loss(outs_test['A2'],y_test)

                grads = self.backward(outs,X_feed,y_feed)
                self.SGD(grads)
```

```
            self.mse_train[f"Epoch:{epoch}"] = loss['MSE']
            self.mce_train[f"Epoch:{epoch}"] = loss['MCE']
            self.mse_test[f"Epoch:{epoch}"] = loss_test['MSE']
            self.mce_test[f"Epoch:{epoch}"] = loss_test['MCE']

            if verbose:
                print(f"[{epoch}/{epochs}] ------> Training :MSE:
{loss['MSE']} and MCE: {loss['MCE']}")
                print(f"[{epoch}/{epochs}] ------> Testing :MSE:
{loss_test['MSE']} and MCE: {loss_test['MCE']}")



    def parameters(self):
        return {'Train_MSE' : self.mse_train,
                'Train_MCE' : self.mce_train,
                'Test_MSE'  : self.mse_test,
                'Test_MCE'  : self.mce_test}




# %%
initialize = 'Xavier'
input_size = X_train.shape
batch_size = 18
hidden_neurons = 19
epochs = 200

model = TwoLayerNetwork(input_size,batch_size,hidden_neurons,lr=1e-1)


# %%
model.fit(X_train,y_train,X_test,y_test,epochs)


# %%
net_params = model.parameters()


# %%
plt.rcParams['figure.figsize'] = (9,6)
plt.plot(net_params['Train_MCE'].values())
plt.xlabel('# of Epoch')
plt.ylabel('Mean Classification Error')
plt.title('MCE versus Epoch in Training')
plt.show()


# %%
plt.plot(net_params['Train_MSE'].values(),color = 'green')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Squared Error')
plt.title('MSE versus Epoch in Training')
plt.show()


# %%
plt.plot(net_params['Test_MCE'].values(),color = 'orange')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Classification Error')
```

```
plt.title('MCE versus Epoch in Validation')
plt.show()


# %%
plt.plot(net_params['Test_MSE'].values(),color = 'blue')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Squared Error')
plt.title('MSE versus Epoch in Validation')
plt.show()

# %% [markdown]
# Part C

# %%
low_hidden_model = TwoLayerNetwork(input_size,batch_size,n_neurons=8,lr=1e-
1)
high_hidden_model =
TwoLayerNetwork(input_size,batch_size,n_neurons=152,lr=1e-1)


# %%
low_hidden_model.fit(X_train,y_train,X_test,y_test,epochs = 200, verbose =
False)
high_hidden_model.fit(X_train,y_train,X_test,y_test,epochs =200, verbose =
False)


# %%
low_hidden_params = low_hidden_model.parameters()
high_hidden_params = high_hidden_model.parameters()


# %%
plt.plot(low_hidden_params['Train_MSE'].values())
plt.plot(net_params['Train_MSE'].values(),color = 'green')
plt.plot(high_hidden_params['Train_MSE'].values(),color = 'orange')
plt.legend(['Low','Optimal','High'])
plt.xlabel('# of Epoch')
plt.ylabel('Mean Squared Error')
plt.title('MSE versus Epoch in Training')
plt.show()


# %%
plt.plot(low_hidden_params['Train_MCE'].values())
plt.plot(net_params['Train_MCE'].values(),color = 'green')
plt.plot(high_hidden_params['Train_MCE'].values(),color = 'orange')
plt.legend(['Low','Optimal','High'])
plt.xlabel('# of Epoch')
plt.ylabel('Mean Classification Error')
plt.title('MCE versus Epoch in Training')
plt.show()


# %%
plt.plot(low_hidden_params['Test_MSE'].values())
plt.plot(net_params['Test_MSE'].values(),color = 'green')
plt.plot(high_hidden_params['Test_MSE'].values(),color = 'orange')
plt.legend(['Low','Optimal','High'])
plt.xlabel('# of Epoch')
```

```
plt.ylabel('Mean Squared Error')
plt.title('MSE versus Epoch in Validation')
plt.show()


# %%
plt.plot(low_hidden_params['Test_MCE'].values())
plt.plot(net_params['Test_MCE'].values(),color = 'green')
plt.plot(high_hidden_params['Test_MCE'].values(),color = 'orange')
plt.legend(['Low','Optimal','High'])
plt.show()

# %% [markdown]
# Part D

# %%
class ThreeLayerNetwork:

    def __init__(self,input_size,batch_size,h1_neurons,h2_neurons, mean =
0,std = 1,lr =1e-1,distribution = 'Xavier'):
        np.random.seed(15)
        self.lr = lr
        self.mse_train = {}
        self.mce_train = {}
        self.mse_test = {}
        self.mce_test = {}
        self.prev_updates = {'W1':0,'W2':0,'W3':0,
                             'B1':0,'B2':0,'B3':0}

        self.h1_neurons = h1_neurons
        self.h2_neurons = h2_neurons

        self.sample_size = input_size[0]
        self.feature_size = input_size[1]
        self.batch_size = batch_size
        self.mean,self.std = mean,std

        self.dist = distribution


        self.n_update = round((self.sample_size/self.batch_size))

        self.W1_size = self.feature_size,self.h1_neurons
        self.W2_size = self.h1_neurons,self.h2_neurons
        self.W3_size = self.h2_neurons,1

        self.B1_size = 1,h1_neurons
        self.B2_size = 1,h2_neurons
        self.B3_size = 1,1




        if (self.dist == 'Zero') :
            self.W1 = np.zeros((self.W1_size))
            self.W2 = np.zeros((self.W2_size))
            self.W3 = np.zeros((self.W3_size))
            self.B1 = np.zeros((self.B1_size))
            self.B2 = np.zeros((self.B2_size))
            self.B3 = np.zeros((self.B3_size))
```

```
        elif (self.dist == 'Gauss'):
            self.W1 = Gauss(loc = self.mean, scale = self.std, size =
(self.W1_size)) * 0.01
            self.W2 = Gauss(loc = self.mean, scale = self.std, size =
(self.W2_size)) * 0.01
            self.W3 = Gauss(loc = self.mean, scale = self.std, size =
(self.W3_size)) * 0.01

            self.B1 = Gauss(loc = self.mean, scale = self.std, size =
(self.B1_size)) * 0.01
            self.B2 = Gauss(loc = self.mean, scale = self.std, size =
(self.B2_size)) * 0.01
            self.B3 = Gauss(loc = self.mean, scale = self.std, size =
(self.B3_size)) * 0.01

        elif (self.dist == 'He'):
            self.he_scale1 = np.sqrt(2/self.feature_size)
            self.he_scale2 = np.sqrt(2/self.h1_neurons)
            self.he_scale3 = np.sqrt(2/self.h2_neurons)

            self.W1 = randn(self.W1_size[0],self.W1_size[1]) *
self.he_scale1
            self.W2 = randn(self.W2_size[0],self.W2_size[1]) *
self.he_scale2
            self.W3 = randn(self.W2_size[0],self.W2_size[1]) *
self.he_scale3

            self.B1 = randn(self.B1_size[0],self.B1_size[1]) *
self.he_scale1
            self.B2 = randn(self.B2_size[0],self.B2_size[1]) *
self.he_scale2
            self.B3 = randn(self.B3_size[0],self.B3_size[1]) *
self.he_scale3

        elif (self.dist == 'Xavier'):
            self.xavier_scale1 =
np.sqrt(2/(self.feature_size+self.h1_neurons))
            self.xavier_scale2 =
np.sqrt(2/(self.h1_neurons+self.h2_neurons))
            self.xavier_scale3 = np.sqrt(2/(self.h2_neurons+1))

            self.W1 = randn(self.W1_size[0],self.W1_size[1]) *
self.xavier_scale1
            self.W2 = randn(self.W2_size[0],self.W2_size[1]) *
self.xavier_scale2
            self.W3 = randn(self.W3_size[0],self.W3_size[1]) *
self.xavier_scale3

            self.B1 = randn(self.B1_size[0],self.B1_size[1]) *
self.xavier_scale1
            self.B2 = randn(self.B2_size[0],self.B2_size[1]) *
self.xavier_scale2
            self.B3 = randn(self.B3_size[0],self.B3_size[1]) *
self.xavier_scale3



    def forward(self,X):

        Z1 = (X @ self.W1)  + self.B1
        A1 = np.tanh(Z1)
```

```
        Z2 = (A1 @ self.W2) + self.B2
        A2 = np.tanh(Z2)
        Z3 = (A2 @ self.W3) + self.B3
        A3 = np.tanh(Z3)

        return {"Z1": Z1,"A1": A1,
                "Z2": Z2,"A2": A2,
                "Z3": Z3,"A3": A3}



    def tanh(self,X):
        return (np.exp(X) - np.exp(-X))/(np.exp(X) + np.exp(-X))

    def tanh_der(self,X):
        return  1 - np.power(np.tanh(X),2)

    def backward(self,outs, X, Y):
        m = self.batch_size

        Z1 = outs['Z1']
        A1 = outs['A1']
        Z2 = outs['Z2']
        A2 = outs['A2']
        Z3 = outs['Z3']
        A3 = outs['A3']


        dZ3 = (A3-Y)* self.tanh_der(Z3)
        dW3 = (1/m) * (A2.T @ dZ3)
        dB3 = (1/m) * np.sum(dZ3, axis=0, keepdims=True)

        dZ2 = np.multiply((dZ3 @ self.W3.T), self.tanh_der(Z2))
        dW2 = (1/m) * (A1.T @ dZ2)
        dB2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)


        dZ1 = np.multiply((dZ2 @ self.W2.T), self.tanh_der(Z1))
        dW1 = (1/m) * (X.T @ dZ1)
        dB1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)


        return {"dW1": dW1, "dW2": dW2,"dW3": dW3,
                "dB1": dB1, "dB2": dB2,"dB3": dB3}

    def Loss(self,pred,y_true):

        mse = np.square(pred-y_true).mean()

        pred[pred>=0]=1
        pred[pred<0]=-1

        mce = (pred == y_true).mean()*100

        return {'MSE':mse,'MCE':mce}


    def SGD(self,grads,momentum = False, mom coeff = None):
        if momentum:
            self.W1 += (-self.lr * grads['dW1'] +  mom_coeff *
self.prev_updates['W1'])
```

```
            self.W2 += (-self.lr * grads['dW2'] +  mom_coeff *
self.prev_updates['W2'])
            self.W3 += (-self.lr * grads['dW3'] +  mom_coeff *
self.prev_updates['W3'])
            self.B1 += (-self.lr * grads['dB1'] +  mom_coeff *
self.prev_updates['B1'])
            self.B2 += (-self.lr * grads['dB2'] +  mom_coeff *
self.prev_updates['B2'])
            self.B3 += (-self.lr * grads['dB3'] +  mom_coeff *
self.prev_updates['B3'])

            self.prev_updates['W1'] = - self.lr * grads['dW1'] +  mom_coeff
* self.prev_updates['W1']
            self.prev_updates['W2'] = - self.lr * grads['dW2'] +  mom_coeff
* self.prev_updates['W2']
            self.prev_updates['W3'] = - self.lr * grads['dW3'] +  mom_coeff
* self.prev_updates['W3']
            self.prev_updates['B1'] = - self.lr * grads['dB1'] +  mom_coeff
* self.prev_updates['B1']
            self.prev_updates['B2'] = - self.lr * grads['dB2'] +  mom_coeff
* self.prev_updates['B2']
            self.prev_updates['B3'] = - self.lr * grads['dB3'] +  mom_coeff
* self.prev_updates['B3']

        else:
            self.W1 -= self.lr * grads['dW1']
            self.W2 -= self.lr * grads['dW2']
            self.W3 -= self.lr * grads['dW3']
            self.B1 -= self.lr * grads['dB1']
            self.B2 -= self.lr * grads['dB2']
            self.B3 -= self.lr * grads['dB3']

    def fit(self,X,Y,X_test,y_test,epochs,momentum = False, mom_coeff =
None,verbose=True):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """



        for epoch in range(epochs):
            perm = np.random.permutation(self.sample_size)
            print(f"Epoch num: {epoch}")
            for idx in range(self.n_update):


                batch_start = idx * self.batch_size
                batch_finish =(idx+1) * self.batch_size
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]


                outs = self.forward(X_feed)
                loss = self.Loss(outs['A3'],y feed)

                outs_test = self.forward(X_test)
                loss_test = self.Loss(outs_test['A3'],y_test)
```

```
                grads = self.backward(outs,X_feed,y_feed)
                self.SGD(grads, momentum = momentum, mom_coeff = mom_coeff)

            self.mse_train[f"Epoch:{epoch}"] = loss['MSE']
            self.mce_train[f"Epoch:{epoch}"] = loss['MCE']
            self.mse_test[f"Epoch:{epoch}"] = loss_test['MSE']
            self.mce_test[f"Epoch:{epoch}"] = loss_test['MCE']

            if verbose:
                print(f"[{epoch}/{epochs}] ------> Training :MSE:
{loss['MSE']} and MCE: {loss['MCE']}")
                print(f"[{epoch}/{epochs}] ------> Testing  :MSE:
{loss_test['MSE']} and MCE: {loss_test['MCE']}")



    def parameters(self):
        return {'Train_MSE' : self.mse_train,
                'Test_MSE'  : self.mse_test,
                'Train_MCE' : self.mce_train,
                'Test_MCE'  : self.mce_test}




# %%
mean = 0.0
std = 0.1
initialize = 'Xavier'
input_size = X_train.shape
batch_size = 18
h1_neurons = 76
h2_neurons = 380
learning_rate = 1e-1
epochs = 200

model2 = ThreeLayerNetwork(input_size,batch_size,h1_neurons,h2_neurons)


# %%
model2.fit(X_train,y_train,X_test,y_test,epochs)


# %%
net_params2 = model2.parameters()


# %%
plt.plot(net_params2['Train_MCE'].values())
plt.xlabel('# of Epoch')
plt.ylabel('Mean Classification Error')
plt.title('MCE versus Epoch in Training')
plt.show()


# %%
plt.plot(net_params2['Train_MSE'].values(),color = 'green')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Squared Error')
plt.title('MSE versus Epoch in Training')
```

```
plt.show()


# %%
plt.plot(net_params2['Test_MCE'].values(),color = 'orange')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Classification Error')
plt.title('MCE versus Epoch in Validation')
plt.show()


# %%
plt.plot(net_params2['Test_MSE'].values())
plt.xlabel('# of Epoch')
plt.ylabel('Mean Squared Error')
plt.title('MSE versus Epoch in Validation')
plt.show()

# %% [markdown]
# Part D

# %%
batch_size = 18
h1_neurons = 38
h2_neurons = 380
learning_rate = 1e-1
epochs = 200

with_mom_model =
ThreeLayerNetwork(input_size,batch_size,h1_neurons,h2_neurons)


# %%
with_mom_model.fit(X_train,y_train,X_test,y_test,epochs, momentum = True,
mom_coeff = 0.1)


# %%
net_params3 = with_mom_model.parameters()


# %%
plt.plot(net_params3['Train_MCE'].values(),color = 'orange')
plt.plot(net_params2['Train_MCE'].values(),color = 'green')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Classification Error')
plt.title('MCE versus Epoch in Training')
plt.legend(['w/ Momentum', 'w/o Momentum'])
plt.show()


# %%
plt.plot(net_params3['Train_MSE'].values(),color = 'orange')
plt.plot(net_params2['Train_MSE'].values(),color = 'green')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Squared Error')
plt.title('MSE versus Epoch in Training')
plt.legend(['w/ Momentum', 'w/o Momentum'])
plt.show()
```

```
# %%
plt.plot(net_params3['Test_MCE'].values(),color = 'orange')
plt.plot(net_params2['Test_MCE'].values(),color = 'green')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Classification Error')
plt.title('MCE versus Epoch in Validation')
plt.legend(['w/ Momentum', 'w/o Momentum'])
plt.show()


# %%
plt.plot(net_params3['Test_MSE'].values(),color = 'orange')
plt.plot(net_params2['Test_MSE'].values(),color = 'green')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Squared Error')
plt.title('MSE versus Epoch in Validation')
plt.legend(['w/ Momentum', 'w/o Momentum'])
plt.show()


# %%



# %%
```

4.2 Q2 Code

```
# To add a new cell, type '# %%'
# To add a new markdown cell, type '# %% [markdown]'
# %%
from IPython import get_ipython

# %%
import numpy as np
from numpy.random import normal as Gauss
import matplotlib.pyplot as plt
import h5py
get_ipython().run_line_magic('matplotlib', 'inline')
from numpy.random import randn


# %%
class DatasetLoader:
    def __init__(self, path):
        self.path = path

    def load(self):
        with h5py.File(self.path,'r') as F:
            names = list(F.keys())
            X_train = np.array(F[names[3]].value)
            y_train = np.array(F[names[2]].value)
            X_test = np.array(F[names[1]].value)
            y_test = np.array(F[names[0]].value)
            X_val = np.array(F[names[5]].value)
            y_val = np.array(F[names[4]].value)
            words = np.array(F[names[-1]].value)
```

```
            return {'TrainX': X_train, 'TrainY' : y_train,
                    'TestX' : X_test,  'TestY'  : y_test,
                    'ValX'  : X_val,   'ValY'   : y_val,
                    'Corpus': words}

path = 'assign2_data2.h5'
data = DatasetLoader(path).load()


# %%
class OneHotEncode:
    def __init__(self):
        self.vocab_size = 250

    def vectorize2D(self,Y):
        size = Y.shape[0]
        val = np.zeros((size,self.vocab_size))

        for i in range(size):
            val[i,int(Y[i])-1]=1
        return val

    def vectorize3D(self,X):
        size = X.shape[0]
        val1 = np.zeros((size,3,self.vocab_size))

        for i in range(size):
            for j in range(3):
                out = np.zeros(self.vocab_size)
                out[X[i,j]-1] = 1
                val1[i,j,:] = out
        return val1

    def transform(self,data,vector):
        if vector == '3D':
            return  self.vectorize3D(data)
        else:
            return  self.vectorize2D(data)


# %%
OneHotEncoder = OneHotEncode()
data['EncodedTrainY'] = OneHotEncoder.transform(data['TrainY'], vector =
'2D')
data['EncodedTestY'] = OneHotEncoder.transform(data['TestY'],vector = '2D')
data['EncodedValY'] = OneHotEncoder.transform(data['ValY'], vector = '2D')

data['EncodedTrainX'] = OneHotEncoder.transform(data['TrainX'], vector =
'3D')
data['EncodedTestX'] = OneHotEncoder.transform(data['TestX'],vector = '3D')
data['EncodedValX'] = OneHotEncoder.transform(data['ValX'], vector = '3D')


# %%
print(f"One hot encode training data dimension     :
{data['EncodedTrainX'].shape}")
print(f"One hot encode training label dimension    :
{data['EncodedTrainY'].shape}")
print(f"One hot encode validation data  dimension  :
{data['EncodedValX'].shape}")
```

```
print(f"One hot encode validation label  dimension :
{data['EncodedValY'].shape}")
print(f"One hot encode testing data  dimension     :
{data['EncodedTestX'].shape}")
print(f"One hot encode testing label  dimension    :
{data['EncodedTestY'].shape}")


# %%
train_data =
np.sum((data['EncodedTrainX'][:,0,:],data['EncodedTrainX'][:,1,:],data['Enc
odedTrainX'][:,2,:]),axis=0)
train_label = data['EncodedTrainY']
val_data =
np.sum((data['EncodedValX'][:,0,:],data['EncodedValX'][:,1,:],data['Encoded
ValX'][:,2,:]),axis=0)
val_label = data['EncodedValY']
test_data =
np.sum((data['EncodedTestX'][:,0,:],data['EncodedTestX'][:,1,:],data['Encod
edTestX'][:,2,:]),axis=0)
test_label = data['EncodedTestY']
emded_size = 32
hidden_size = 256


# %%
train_data_conc =
np.concatenate((data['EncodedTrainX'][:,0,:],data['EncodedTrainX'][:,1,:],d
ata['EncodedTrainX'][:,2,:]),axis=1)
train_label_conc = data['EncodedTrainY']
val_data_conc =
np.concatenate((data['EncodedValX'][:,0,:],data['EncodedValX'][:,1,:],data[
'EncodedValX'][:,2,:]),axis=1)
val_label_conc = data['EncodedValY']
test_data_conc =
np.concatenate((data['EncodedTestX'][:,0,:],data['EncodedTestX'][:,1,:],dat
a['EncodedTestX'][:,2,:]),axis=1)
test_label_conc = data['EncodedTestY']


# %%
print(f"One hot encode training data dimension     :
{train_data_conc.shape}")
print(f"One hot encode val label dimension         :
{train_data_conc.shape}")


# %%
def do_not_compute():
    best_model = None
    best_val = -1
    lr = [0.01,0.05,0.1,0.15,0.3,0.5,0.1,0.17]
    batch_size = [50,100,150,200,250,150,250,100]
    alpha = [0.1,0.3,.5,0.7,0.85,0.9,0.6,0.4]
    acc = []
    for i in range(8):
        model =
Word2Vec(embed size=32,hidden size=256,lr=lr[i],batch size=batch size[i],al
pha=alpha[i])
        #model.fit(X_feed,data['TrainY'],30)
        preds = model.predict(X_test); labels = np.argmax(data['TestY'],1)
```

```
        val_acc = accuracy(preds,labels)
        acc.append(val_acc)

        if best_val < val_acc :
            best_val = val_acc
            best_model = model


# %%
class Word2Vec:

    def __init__(self,embed_size,hidden_size):
        np.random.seed(42)
        self.lr = 0.15
        self.batch_size = 250
        self.alpha = 0.85

        self.loss_train_list = []
        self.loss_test_list = []
        self.acc_train_list = []
        self.acc_test_list = []

        self.prev_updates = {'WE':0,'W1':0,'W2':0,
                             'B1':0,'B2':0}

        self.D = embed_size
        self.P = hidden_size

        self.sample_size = data['TrainX'].shape[0]
        self.feature_size = data['TrainX'].shape[1]
        self.vocab_size = data['Corpus'].shape[0]

        self.n_update = round((self.sample_size/self.batch_size))

        self.W_emb_size = self.vocab_size,self.D
        self.W1_size = self.D,self.P
        self.W2_size = self.P,self.vocab_size
        self.B1_size = 1,self.P
        self.B2_size = 1,self.vocab_size

        self.WE =  Gauss(0,scale = 0.01, size = (self.W_emb_size))
        self.W1 = Gauss(0,scale = 0.01, size = (self.W1_size))
        self.W2 = Gauss(0,scale = 0.01, size = (self.W2_size))
        self.B1 = Gauss(0,scale = 0.01, size = (self.B1_size))
        self.B2 = Gauss(0,scale = 0.01, size = (self.B2_size))

    def sigmoid(self,X):
        return 1/(1 + np.exp(-X))


    def MinMaxScaling(self,X):
        return (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

    def softmax_stable(self,X):
        e_x = np.exp(X - np.max(X, axis=-1, keepdims=True))
        return e_x / np.sum(e_x, axis=-1, keepdims=True)


    def log_softmax_stable(self,x):
        e_x = np.exp(x - np.max(x))
        return np.log(e_x / np.sum(x,axis=0))
```

```
    def log_softmax(self,x):
        x = x - np.max(x)
        a = np.logaddexp.reduce(x)
        return np.exp(x - a)

    def CrossEntropyLoss(self,pred,label):
        m = pred.shape[0]

        preds = np.clip(pred, 1e-16, 1 - 1e-16)
        loss = np.sum(-label * np.log(preds) - (1 - label) * np.log(1 -
preds))
        return loss/m


    def linear(self,inp,W):
        return np.dot(inp,W)

    def forward(self,X):
        emb_linear = self.linear(X,self.WE)

        Z1 = self.linear(emb_linear,self.W1) - self.B1

        A1 = self.sigmoid(Z1)
        Z2 = self.linear(A1,self.W2) - self.B2

        A2 = self.softmax_stable(Z2)

        return {"Z1": Z1,"A1": A1,
                "Z2": Z2,"A2": A2,
                "E" : emb_linear}

    def sigmoid_gradient(self,X):
        return self.sigmoid(X) * (1-self.sigmoid(X))

    def softmax_gradient_dZ(self,A2,Y):
        return A2-Y

    def linear_gradients(self,inp,delta):
        return { 'dW' : self.linear(inp,delta)/self.batch_size,
                 'dB' : np.sum(delta, axis=0,
keepdims=True)/self.batch_size}


    def cross_entropy_gradient(self, preds, label):
        preds = np.clip(preds, 1e-15, 1 - 1e-15)
        grad_ce = - (label/preds) + (1 - label) / (1 - preds)
        return grad_ce

    def softmax_stable_gradient(self, X):
        soft_out = self.softmax_stable(X)
        return soft_out * (1 - soft_out)

    def backward(self,outs, X, Y):

        E  = outs['E']
        Z1 = outs['Z1']
        A1 = outs['A1']
        Z2 = outs['Z2']
        A2 = outs['A2']
```

```
        dZ2 = self.cross_entropy_gradient(A2,Y) *
self.softmax_stable_gradient(Z2)
        dW2 = self.linear_gradients(A1.T,dZ2)['dW']
        dB2 = self.linear_gradients(A1.T,dZ2)['dB']

        dZ1 = self.linear(dZ2,self.W2.T) * self.sigmoid_gradient(Z1)
        dW1 = self.linear_gradients(E.T ,dZ1)['dW']
        dB1 = self.linear_gradients(E.T ,dZ1)['dB']

        dEmbed = self.linear(dZ1,self.W1.T)
        dWE = self.linear_gradients(X.T,dEmbed)['dW']

        return {"dW1": dW1, "dW2": dW2,
                "dB1": dB1, "dB2": dB2,
                "dWE": dWE}

    def SGD(self,grads):
        delta_E  = -self.lr * grads['dWE'] +  self.alpha *
self.prev_updates['WE']
        delta_W1 = -self.lr * grads['dW1'] +  self.alpha *
self.prev_updates['W1']
        delta_W2 = -self.lr * grads['dW2'] +  self.alpha *
self.prev_updates['W2']
        delta_B1 = -self.lr * grads['dB1'] +  self.alpha *
self.prev_updates['B1']
        delta_B2 = -self.lr * grads['dB2'] +  self.alpha *
self.prev_updates['B2']

        self.WE += delta_E
        self.W1 += delta_W1
        self.W2 += delta_W2
        self.B1 += delta_B1
        self.B2 += delta_B2


        self.prev_updates['WE'] = delta_E
        self.prev_updates['W1'] = delta_W1
        self.prev_updates['W2'] = delta_W2
        self.prev_updates['B1'] = delta_B1
        self.prev_updates['B2'] = delta_B2

        pass
    def fit(self,X,Y,X_val,y_val,epochs,verbose=True, crossVal = False):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """


        for epoch in range(epochs):
            perm = np.random.permutation(self.sample_size)
            print(f'Epoch : {epoch + 1}')

            for i in range(self.n_update):

                batch start  =  i * self.batch size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]
                X_feed = X[index]
```

```
                y_feed = Y[index]

                outs_train = self.forward(X_feed)
                grads = self.backward(outs_train,X_feed,y_feed)
                self.SGD(grads)

                if crossVal:
                    stop = self.cross_validation(X,val_X,Y,val_Y,threshold
= 5)
                    if stop:
                        break

            cross_loss_train =
self.CrossEntropyLoss(outs_train['A2'],y_feed)
            predictions_train = self.predict(X)
            acc_train =
self.accuracy_score(predictions_train,np.argmax(Y,1))

            cross_loss_val = self.CrossEntropyLoss(X_val,y_val)
            predictions_val = self.predict(X_val)
            acc_val =
self.accuracy_score(predictions_val,np.argmax(y_val,1))

            if verbose:
                print(f"[{epoch + 1}/{epochs}] ------> Training :
Accuracy: {acc_train}")
                print(f"[{epoch + 1}/{epochs}] ------> Testing  :
Accuracy: {acc_val}")

            self.loss_train_list.append(cross_loss_train)
            self.loss_test_list.append(cross_loss_val)
            self.acc_train_list.append(acc_train)
            self.acc_test_list.append(acc_val)


    def
cross_validation(self,train_data,val_data,label_train,label_val,threshold):
        train_preds = self.predict(train_data)
        val_preds = self.predict(val_data)
        train_loss = self.CrossEntropyLoss(train_preds,label_train)
        val_loss = self.CrossEntropyLoss(val_preds,label_val)

        if train_loss - val_loss < threshold:
            return True
        return False

    def top_10_predictions(self,preds):
        return np.argpartition(preds, -10)[-10:]


    def predict(self,X):
        feed = self.forward(X)
        return np.argmax(feed['A2'],axis=1)


    def accuracy_score(self,preds,label):
        expand = 100
        count = 0
        size = label.shape[0]
        for i in range(size):
            if preds[i] == label[i]:
```

```
                count +=1
        return  expand * (count/size)

    def history(self):
        return {'TrainLoss'    : self.loss_train_list,
                'ValLoss'      : self.loss_test_list,
                'TrainAcc'     : self.acc_train_list,
                'ValAcc'       : self.acc_test_list}


# %%
model = Word2Vec(32,256)
model.fit(train_data,train_label,test_data,test_label,50)


# %%
model_history = model.history()


# %%
plt.plot(model_history['TrainLoss'])
plt.xlabel('# of Epoch')
plt.title('Training Cross-Entropy')
plt.legend(['(D,P) = (32,256)'])
plt.show()


# %%
plt.plot(model_history['TrainAcc'],color = 'green')
plt.xlabel('# of Epoch')
plt.title(' Training Accuracy')
plt.legend(['(D,P) = (32,256)'])
plt.show()


# %%
plt.plot(model_history['ValAcc'],color = 'orange')
plt.xlabel('# of Epoch')
plt.title(' Validation Accuracy')
plt.legend(['(D,P) = (32,256)'])
plt.show()


# %%
# Note that np.random.seed() is not placed to see different results in
differents runnings
test_preds = model.predict(test_data)
test_acc = model.accuracy_score(test_preds,np.argmax(test_label,1))
print(f'Test Accuracy: {test_acc}')

forward = model.forward(test_data)
probs_softmax = forward['A2']

num_sample = test_data.shape[0]
n_predict = 5
random_idx = np.random.randint(num_sample,size = n_predict)

# 5 random sample's probabilities
five_random_num_probs = probs_softmax[random_idx]
random_five_words = data['Corpus'][data['TestX'][random_idx]].astype('U13')
```

```
# %%
def top_k_preds(probs,k,n_predict):
    top_k = np.zeros((n_predict,k))
    for i in range(n_predict):
         top_k[i] = probs.argsort()[i][-k:][::-1]
    return top_k
top_10_preds = top_k_preds(five_random_num_probs,10,5)
top_10_words = data['Corpus'][top_10_preds.astype('int')].astype('U13')


# %%
input_words = [list(random_five_words[i]) for i in range(n_predict)]
preds_words = [list(top_10_words[i]) for i in range(n_predict)]
print('_____________________________________________________________________
_____________________\n')
for i in range(n_predict):
    print('Input word',i+1,'----->',  ', '.join(input_words[i]),'\nTop 10
predicted candicates :', ', '.join(preds_words[i]),
'\n_____________________________________________________________________
_____________________\n')


# %%
#model_16x128 = Word2Vec(16,128)
#model_16x128.fit(train_data,train_label,test_data,test_label,epochs = 50)


# %%
#model_16x128_history = model_16x128.history()


# %%
#plt.plot(model_16x128_history['TrainLoss'])
#plt.legend(['(D,P) = (16,128)'])
#plt.xlabel('# of Epoch')
#plt.title('Training Cross-Entropy')
#plt.show()


# %%
#plt.plot(model_16x128_history['TrainAcc'],color = 'green')
#plt.legend(['(D,P) = (16,128)'])
#plt.xlabel('# of Epoch')
#plt.title(' Training Accuracy')
#plt.show()


# %%
#plt.plot(model_16x128_history['ValAcc'],color = 'orange')
#plt.legend(['(D,P) = (16,128)'])
#plt.xlabel('# of Epoch')
#plt.title('Validation Accuracy')
#plt.show()


# %%
#model_8x64 = Word2Vec(8,64)
#model_8x64.fit(train_data,train_label,test_data,test_label,epochs = 50)
```

```
# %%
#model_8x64_history = model_8x64.history()


# %%
#plt.plot(model_8x64_history['TrainLoss'])
#plt.legend(['(D,P) = (8,64)'])
#plt.xlabel('# of Epoch')
#plt.title('Training Cross-Entropy')
#plt.show()


# %%
#plt.plot(model_8x64_history['TrainAcc'],color = 'green')
#plt.legend(['(D,P) = (8,64)'])
#plt.xlabel('# of Epoch')
#plt.title(' Training Accuracy')
#plt.show()


# %%
#plt.plot(model_8x64_history['ValAcc'],color = 'orange')
#plt.legend(['(D,P) = (8,64)'])
#plt.xlabel('# of Epoch')
#plt.title('Validation Accuracy')
#plt.show()
```

5 REFERENCES

[1] X. Rong, “word2vec Parameter Learning Explained,” Jun. 2016.

[2] A. Jana, A. S. says, and A. Sharma, “Understanding and implementing Neural Network with SoftMax in Python from scratch,” *A Developer Diary*, 29-Apr-2019. [Online]. Available: https://www.adeveloperdiary.com/data-science/deep-learning/neural-network-with-softmax-in-python/. [Accessed: 17-Nov-2020].

[3] D. Karani, “Introduction to Word Embedding and Word2Vec,” *Medium*, 02-Sep-2020. [Online]. Available: https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa. [Accessed: 17-Nov-2020].

[4] I. Chen, “Word2vec from Scratch with NumPy,” *Medium*, 18-Feb-2019. [Online]. Available: https://towardsdatascience.com/word2vec-from-scratch-with-numpy-8786ddd49e72. [Accessed: 17-Nov-2020].

[5] L. N. Wijayasingha, “Are You Messing With Me Softmax?,” *Medium*, 13-Aug-2020. [Online]. Available: https://medium.com/swlh/are-you-messing-with-me-softmax-84397b19f399. [Accessed: 17-Nov-2020].

[6] “Implement your own word2vec(skip-gram) model in Python,” *GeeksforGeeks*, 21-Jan-2019. [Online]. Available: https://www.geeksforgeeks.org/implement-your-own-word2vecskip-gram-model-in-python/?ref=lbp. [Accessed: 17-Nov-2020].

[7] Rahuljha, “Word2Vec implementation,” *Medium*, 29-Jun-2020. [Online]. Available: https://towardsdatascience.com/a-word2vec-implementation-using-numpy-and-python-d256cf0e5f28. [Accessed: 17-Nov-2020].

[8] “Python: Word Embedding using Word2Vec,” *GeeksforGeeks*, 18-May-2018. [Online]. Available: https://www.geeksforgeeks.org/python-word-embedding-using-word2vec/. [Accessed: 17-Nov-2020].

[9] W. Arliss, “Why and How to use Cross Entropy,” *Medium*, 22-Oct-2020. [Online]. Available: https://towardsdatascience.com/why-and-how-to-use-cross-entropy-4e983cbdd873. [Accessed: 17-Nov-2020].