EEE443 - NEURAL NETWORKS

HOMEWORK ASSIGNMENT - 2

Berkan Ozdamar

21602353

Question 1

In this question, we are given a data of images which contains cat and car images. We are asked to apply stochastic gradient descent on mini bathes. Then mean square error and mean classification error is used as the error metrics.

Part a

We are asked to design a multi-layer neural network with 1 hidden layer. The learning rate $\eta \in [0.1, 0.5]$, number of neurons N in the hidden layer, mini-batch size and the initialization of weights and biases are experimented. After selecting the parameters, backpropagation is applied until convergence. Then, training squared error, testing squared error, training classification error, and testing classification error are plotted versus each epoch. Also, a hyperbolic tangent(tanh) activation function is used for all neurons.

First, we can define MSE as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

And, the hyperbolic tangent(tanh) is:

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

The sizes of training data, training labels, test data and test labels are given below.

[1]:
```
import numpy as np
import h5py
import matplotlib.pyplot as plt
```

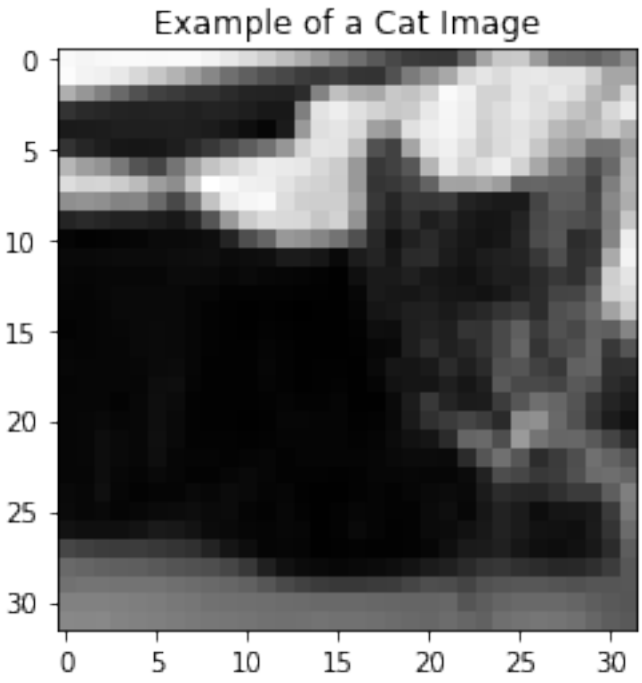
[2]:
```
# Part A

f = h5py.File('assign2_data1.h5', 'r')
dataKeys = list(f.keys())
print('The data keys are:' + str(dataKeys))

# Gathering the  train images, test images, train labels and test labels.
testims = f['testims']
testlbls = f['testlbls']
trainims = f['trainims']
trainlbls = f['trainlbls']
print('The size of testims is: ' + str(np.shape(testims)))
print('The size of testlbls is: ' + str(np.shape(testlbls)))
print('The size of trainims is: ' + str(np.shape(trainims)))
print('The size of trainlbls is: ' + str(np.shape(trainlbls)))
```

```
The data keys are:['testims', 'testlbls', 'trainims', 'trainlbls']
The size of testims is: (1000, 32, 32)
The size of testlbls is: (1000,)
The size of trainims is: (1900, 32, 32)
The size of trainlbls is: (1900,)
```

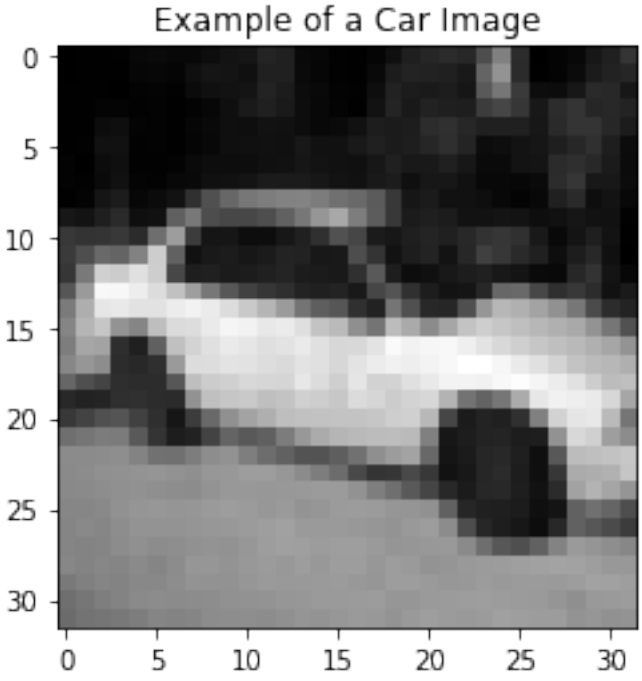
From the training images, an example of a car image and an example of a car image is given below:

[3]:
```
figureNum = 0
plt.figure(figureNum)
plt.title('Example of a Cat Image')
```

```
plt.imshow(trainims[0].T, cmap='gray')
plt.show()
```



[4]:
```
figureNum = 0
plt.figure(figureNum)
plt.title('Example of a Car Image')
plt.imshow(trainims[1500].T, cmap='gray')
plt.show()
```



Then, for the MLP construction, 2 classes are introduced. Those classes are HiddenLayer class and MLP class. In the HiddenLayer class, the corresponding weight updates, bias updates, momentum updates, weights and bias are stored. Initially, weights and bias are random Gaussian distribution with 0 mean and 0.02 standard deviation. Also. in the HiddenLayer class, all the update related variables are initially None and 0.

[5]:
```
class HiddenLayer:

    def __init__(self, neuronNum, neuronSize, mean, std):

        '''
        This class creates a hidden laer for neural network.
        Weights and bias are initially random Gaussian distribution.

        INPUTS:

            neuronNum    : neuronNum is the features a neuron holds.
            neuronSize   : neuronSize is the number of neurons in a hidden layer.
            mean         : mean for Gaussian distribution.
            std          : Standard deviation for Gaussian distribution.

        RETURNS:

        '''

        np.random.seed(15)
        self.weights = np.random.normal( loc=mean, scale=std,␣
↪size=(neuronNum,neuronSize))
        self.bias = np.random.normal( loc=mean, scale=std, size=(1,neuronSize))
        self.Z = None
        self.A = None
        self.grad = None
        self.dB = None
        self.dW = None
        self.error = None
        self.momentum_dw = 0
        self.momentum_db = 0
```

In the MLP class, addLayer function is defined to add hidden layers to the network. Then as activation function tanh is defined as:

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

Also, the derivative of tanh(x) is calculated since it is required when applying back-propagation. The derivative of tanh(x), notated as der_tanh(x), is:

$$\frac{dtanh(x)}{dx} = der\_tanh(x) = sech^2(x) = 1 - tanh^2(x)$$

Moreover, since the loss function in this network is MSE, the derivative of the loss function is also required for back propagation. Hence, MSE and der_MSE can be written as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$der\_MSE = \hat{y} - y$$

The python code is given below as:

[6]:
```
class MLP:
    def __init__(self, momentum = False, momentumCoef = 0):

        '''
        This class creates a multilayer perceptron network.

        INPUTS:

            momentum     : momentum is the boolean for the network which indicates
                           whether the momentum learning will be done or not

            momentumCoef : Coefficient of momentum learning


        RETURNS:

        '''

        self.momentum = momentum
        self.momentumCoef = momentumCoef
        self.layers = list()
        self.batchSize = 0


    def addLayer(self, layer):
        '''
        This function adds a HiddenLayer class to the network.

        INPUTS:

            layer        : layer is an instance of HiddenLayer class

        RETURNS:

        '''

        self.layers.append(layer)

    def tanh(self, x):
        '''
        This function is the hyperbolic tangent for the activation functions of␣
↪each neuron.
        INPUTS:

            x            : x is the weighted sum which will be pushed to activation␣
↪function.

        RETURNS:

            result       : result is the hyperbolic tangent of the input x.
        '''
```

```
        result = 2 / (1 + np.exp(-2*x)) - 1
        return result

    def der_tanh(self, x):
        '''
        This function is the derivative hyperbolic tangent. This function will be␣
↪used in backpropagation.
        INPUTS:

            x             : x is the input.

        RETURNS:

            result        : result is the derivative of hyperbolic tangent of the␣
↪input x.
        '''
        result = 1 - self.tanh(x)**2
        return result

    def MSE(self, y, y_pred):
        '''
        MSE is the loss function for the network.
        INPUTS:

            y             : y is the labels for our data.
            y_pred        : y_pred is the network's prediction.

        RETURNS:

            loss          : loss is the mean squared error between y and y_pred.
        '''
        error = y - y_pred
        loss = np.mean(error**2)
        return loss

    def der_MSE(self, y, y_pred):
        '''
        der_MSE is the derivative of loss function for the network.
        This function will be used for backpropagation.
        INPUTS:

            y             : y is the labels for our data.
            y_pred        : y_pred is the network's prediction.

        RETURNS:

            result        : result is the derivative of the MSE between y and y_pred.
        '''
        result = y_pred - y
        return result

    def MCE(self, y, y_pred):
        '''
        MCE is the accuracy of our network. Mean classification error will be␣
↪calculated to find accuracy.
        INPUTS:
```

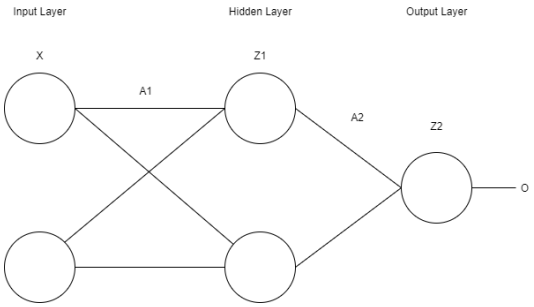
```
        y             : y is the labels for our data.
        y_pred        : y_pred is the network's prediction.

    RETURNS:

                      : returns the accuracy between y and y_pred.
    '''
    count = 0
    for i in range(len(y)):
        if(y[i] == y_pred[i]):
            count += 1
    return 100 * (count / len(y))
```

Here, the activation and loss function and their derivatives are defined. So, the forward propagation, back-propagation and weights updates will be calculated. First, introduce the notations that will be used:

W_n : Weights matrix for the n^{th} layer.
B_n : Bias vector for the n^{th} layer.
A_n : $A_n = W_n^T X + B_n$, where X is the input to n^{th} layer.
Z_n : $Z_n = \phi(A_n)$, where $\phi(.)$ is the activation function of n^{th} layer.
o : MSE(y, $\hat{y}$), which is the MSE of the label y and the predicted output $\hat{y}$

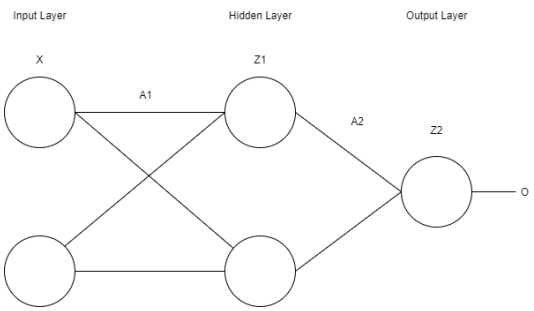
Now, we can explain the forward propagation, back-propagation and weight updates.

Forward Propagation



With inputs from the previous layer, each unit computes the transformation $A_n = W_n^T X + B_n$. Then, the n^{th} layer's activation function is applied to A_n. Hence, we get $Z_n = \phi(A_n)$, where $\phi(.)$ is the tanh(.) in this question for all layers. So, $Z_n = tanh(A_n)$. In this process, Z_n, A_n are stored since those variables are used in back-propagation. Finally the Z_n found in the n^{th} layer, is the input for the $n + 1^{th}$ layer. This procedure is done recursively from the 1^{st} hidden layer to output layer. Z_o which is the Z for output layer, is the $\hat{y}$. Finally, MSE(y, $\hat{y}$) is computed and it is the output o.

Back Propagation



In back propagation, we want to find gradients of weights by taking partial derivatives of $\frac{dO}{dW_n}$. By looking at the figure above, we can say that the $o = MSE(y, \hat{y})$, $\hat{y} = Z_2$, $Z_2 = tanh(A_2)$ and finally $A_2 = W_2^T Z_1 + B_2$ With the help of the figure above, we can represent the $\frac{\partial O}{\partial W_2}$ as:

$$\frac{\partial O}{\partial W_2} = \frac{\partial O}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial A_2} \frac{\partial A_2}{\partial W_2}$$

$$\frac{\partial O}{\partial W_2} = (der\_MSE(\hat{y}) * der\_tanh(A_2))^T Z_1$$

where, $der\_MSE(\hat{y}) = \hat{y} - y$ and $der\_tanh(A_2) = 1 - tanh^2(A_2)$. So:

$$\frac{\partial O}{\partial W_2} = ((\hat{y} - y) * (1 - tanh^2(A_2)))^T Z_1$$

and

$$\frac{\partial O}{\partial B_2} = \frac{\partial O}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial A_2} \frac{\partial A_2}{\partial B_2} = ((\hat{y} - y) * (1 - tanh^2(A_2)))$$

Similarly:

$$\frac{\partial O}{\partial W_1} = \frac{\partial O}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial A_2} \frac{\partial A_2}{\partial Z_1} \frac{\partial Z_1}{\partial A_1} \frac{\partial A_1}{\partial W_1}$$

and

$$\frac{\partial O}{\partial B_1} = \frac{\partial O}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial A_2} \frac{\partial A_2}{\partial Z_1} \frac{\partial Z_1}{\partial A_1} \frac{\partial A_1}{\partial B_1}$$

It can be seen that the $\frac{dO}{d\hat{y}} \frac{d\hat{y}}{dA_2}$ term is also in the gradient of $\frac{dO}{dW_2}$. Thus, this allows us to store those pre-calculated values to update the previous layer's weights and bias. This kind of pattern contains for all previous layer updates and for each layer update this kind of pattern is used iteratively for ease of computation.

Update Weights

After finding the gradients of each layer's weights and bias update, the weight update is simply:

$$W_n = W_n - \eta \frac{\partial O}{\partial W_n}$$

where the η is the learning rate $\in [0.1, 0.5]$.

The python code for implementation of the forward propagation, back-propagation and the update weights function is given below. Also in the update weights function, there is momentum update part but it will be discussed in a later part of this question.

[6]:
```
def forward(self, data):
        '''
        forward function is the forward propagation.
        INPUTS:

            data          : data is the input which will pushed to forward␣
↪propagation.

        RETURNS:

                          : returns the prediction of the network.
        '''

        layerSize = np.shape(self.layers)[0]
```

```
        for i in range(layerSize):
            # 1st Hidden Layer
            if (i == 0):
                self.layers[i].A = data.dot(self.layers[i].weights) + self.
↪layers[i].bias
                self.layers[i].Z = self.tanh(self.layers[i].A)
            # Other Hidden Layers and Output Layer
            else:
                self.layers[i].A = (self.layers[i-1].Z).dot(self.layers[i].weights)␣
↪+ self.layers[i].bias
                self.layers[i].Z = self.tanh(self.layers[i].A)

        return self.layers[-1].Z


    def back_propagation(self, data, label):
        '''
        back_propagation function is the back propagation algorithm for weight and␣
↪bias updates.
        back_propagation function first calls forward function to predict the␣
↪output of network which us y_pred.
        INPUTS:

            data          : data is the input.
            label         : label is the labels of the data.

        RETURNS:

        '''

        layerSize = np.shape(self.layers)[0]
        y_pred = self.forward(data)

        for i in range(layerSize)[::-1]:
            # Output Layer
            if (i == layerSize - 1):
                self.layers[i].error = self.der_MSE(label, y_pred)
                self.layers[i].error = np.array(self.layers[i].error).reshape(-1,1)
                self.layers[i].grad = (self.layers[i].error) * (self.der_tanh(self.
↪layers[i].A))
                self.layers[i].dW = (self.layers[i-1].Z).T.dot(self.layers[i].grad)
                self.layers[i].dB = np.sum(self.layers[i].grad, axis=0,␣
↪keepdims=True)

            # 1st Hidden Layer
            elif(i == 0):
                self.layers[i].error = (self.layers[i+1].grad).dot(self.layers[i+1].
↪weights.T)
                self.layers[i].grad = (self.layers[i].error) * self.der_tanh(self.
↪layers[i].A)
                self.layers[i].dW = data.T.dot(self.layers[i].grad)
                self.layers[i].dB = np.sum(self.layers[i].grad, axis=0,␣
↪keepdims=True)

            # Other Hidden Layers
            else:
```

```
                self.layers[i].error = (self.layers[i+1].grad).dot(self.layers[i+1].
↪weights.T)

                self.layers[i].grad = (self.layers[i].error) * self.der_tanh(self.
↪layers[i].A)
                self.layers[i].dW = (self.layers[i-1].Z).T.dot(self.layers[i].grad)
                self.layers[i].dB = np.sum(self.layers[i].grad, axis=0,␣
↪keepdims=True)


    def update_weights(self, data, label, learningRate):
        '''
        update_weights function updates the weights with the gradients found with␣
↪back_propagation.
        INPUTS:

            data         : data is the input.
            label        : label is the labels of the data.
            learnigRate  : learningRate is the coefficient for the weight update.

        RETURNS:

        '''


        layerSize = np.shape(self.layers)[0]
        self.back_propagation(data,label)
        # If momentum is used.
        if( self.momentum == True ):
            for i in range(layerSize):

                self.layers[i].momentum_dw = self.layers[i].dW + (self.momentumCoef␣
↪* self.layers[i].momentum_dw)
                self.layers[i].momentum_db = self.layers[i].dB + (self.momentumCoef␣
↪* self.layers[i].momentum_db)

                self.layers[i].weights -= (learningRate * self.layers[i].
↪momentum_dw)/self.batchSize
                self.layers[i].bias -= (learningRate * self.layers[i].momentum_db)/
↪self.batchSize

        #If momentum is not used.
        else:
            for i in range(layerSize):

                self.layers[i].weights -= (learningRate * self.layers[i].dW)/self.
↪batchSize
                self.layers[i].bias -= (learningRate * self.layers[i].dB)/self.
↪batchSize
```

Then, the predict function is defined. Since the range of tanh(.) is between -1 and 1, the labels are transformed into 1 and -1 from 1 and 0. Hence, the prediction is done taking 0 as threshold. If the output is smaller than 0 then the prediction is -1, otherwise prediction is 1. The code is given below:

[5]:
```
def predict(self, y_pred):
        '''
        predict function predicts and output from the network's output y_pred.
        INPUTS:

            y_pred       : MLP's output.

        RETURNS:

                         : returns the label for prediction of the network.
        '''

        return np.where(y_pred>=0, 1, -1)
```

Finally, the training function is defined. The mini-batch gradient descent algorithm is used for updating weights. So, the training data is divided into mini-batches and after each mini-batch, a weight update operation is done for all layers. After each epoch, the MSE and MCE for training data, also MSE and MCE for test data is stored. The code is given below:

[5]:
```
   def trainNetwork(self, data, label, testData, testLabel, learningRate,␣
↪batchNum, epoch):
        '''
        trainNetwork function calls the update_weights function to train the␣
↪network over mini-batches
        for fiven number of epochs.
        INPUTS:

            data         : data is the training data.
            label        : label is the labels of the data.
            testData     : testData is the test data.
            testLabel    : testLabel is the labels of the testData.
            learnigRate  : learningRate is the coefficient for the weight update.
            batchNum     : batchNum is the number of mini-batches.
            epoch        : Number of times the network train the whole data.

        RETURNS:

            MSE_loss     : MSE loss of the training data.
            MCE_loss     : MCE loss of the training data.
            test_MSE     : MSE loss of the test data.
            test_MCE     : MCE loss of the test data.
        '''

        MSE_loss = list()
        MCE_loss = list()
        test_MSE = list()
        test_MCE = list()
        np.random.seed(7)

        for i in range(epoch):

            randomIndexes = np.random.permutation(len(label))
            data = data[randomIndexes]
            label = label[randomIndexes]
```

```
            batchLength = len(label) / batchNum
            self.batchSize = batchLength
            for j in range(batchNum):

                start = int(batchLength*j)
                end = int(batchLength*(j+1))
                self.update_weights(data[start:end],label[start:end],learningRate)

            y_pred = self.forward(data)
            loss = self.MSE(label, y_pred)
            MSE_loss.append(loss)


            loss_MCE = self.MCE(label, self.predict(y_pred))
            MCE_loss.append(loss_MCE)

            y_pred_test = self.forward(testData)
            test_loss = self.MSE(testLabel, y_pred_test)
            test_MSE.append(test_loss)


            test_loss_MCE = self.MCE(testLabel, self.predict(y_pred_test))
            test_MCE.append(test_loss_MCE)

        return MSE_loss, MCE_loss, test_MSE, test_MCE
```

Now, the 2 classes HiddenLayer and MLP are explained. Below, the label manipulation (changing the 0 classes into -1 like discussed in predict function.) is done and the image data is flattened.

[7]:
```
trainimages = np.asarray(trainims)
testimages = np.asarray(testims)
trainlabels = np.asarray(trainlbls)
testlabels = np.asarray(testlbls)

trainlabels = np.where(trainlabels == 0 , -1, 1)
testlabels = np.where(testlabels == 0 , -1, 1)

trainlabels = trainlabels.reshape(-1,1)
testlabels = testlabels.reshape(-1,1)

train_img_flat = trainimages.reshape(1900, 32**2)
test_img_flat = testimages.reshape(1000, 32**2)
```

Here, the grid search is done for a range of learning rate and a range of N neuron size. Since the run takes a while, after finding the best learning rate and N values, I have commented the cell and stored those values.

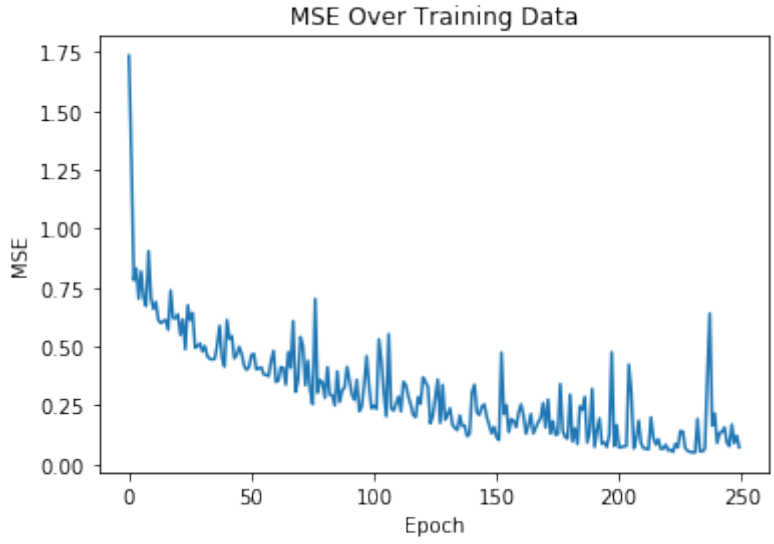
[8]:
```
# learningRate = np.arange(0.1, 0.5, 0.05)
# N_layer = np.arange(10,30,2)

# best_lr = 0
# best_n = 0
# mse_best = np.inf

# for n in N_layer:
#     print(n)
#     neuralNet = MLP()
```

```
#     neuralNet.addLayer(HiddenLayer(32**2, n, 0, 0.02))
#     neuralNet.addLayer(HiddenLayer(n, 1, 0, 0.02))
#     for lr in learningRate:
#         mse_loss, mce_loss, test_mse, test_mce = neuralNet.
 ↪trainNetwork(train_img_flat/255, trainlabels, test_img_flat/255, testlabels, lr,␣
 ↪50, 300)
#         if(mse_loss[-1] < mse_best):
#             best_lr = lr
#             best_n = n
#             mse_best = mse_loss[-1]
```

Below, the MLP with 1 hidden layer is trained with the best parameters found above. The learning rate $\eta = 0.35$ and N = 20.

[10]:
```
best_lr = 0.35
best_n = 20

neuralNet = MLP()
neuralNet.addLayer(HiddenLayer(32**2, best_n, 0, 0.02))
neuralNet.addLayer(HiddenLayer(best_n, 1, 0, 0.02))

mse_loss, mce_loss, test_mse, test_mce = neuralNet.trainNetwork(train_img_flat/255,␣
 ↪trainlabels, test_img_flat/255, testlabels, best_lr, 50, 250)
```

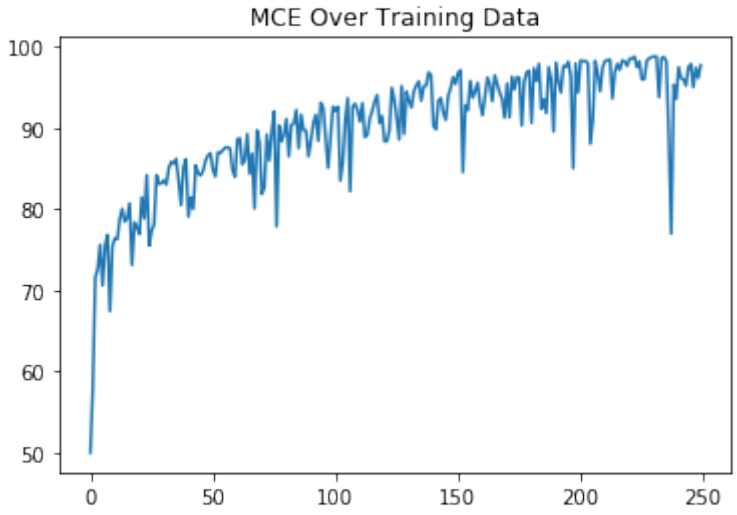
[11]:
```
figureNum = 0
plt.figure(figureNum)
plt.plot(mse_loss)
plt.title('MSE Over Training Data')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.show()
```



Here, it can be seen that the MSE over training data converges to 0. This means that the algorithm starts to memorize the data.

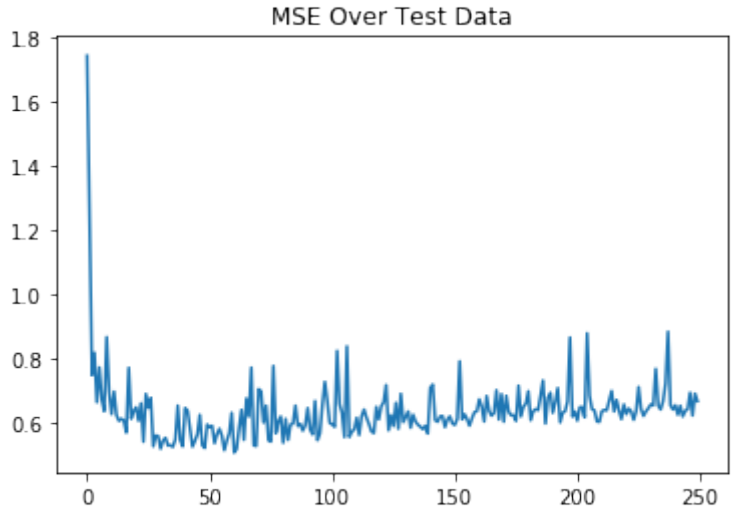
[12]:
```
figureNum += 1
plt.figure(figureNum)
```

```
plt.plot(mce_loss)
plt.title('MCE Over Training Data')
plt.show()
```



Also, classification accuracy of the training data converges to 100. Now, we will inspect the MSE and MCE over the test data and make conclusions.

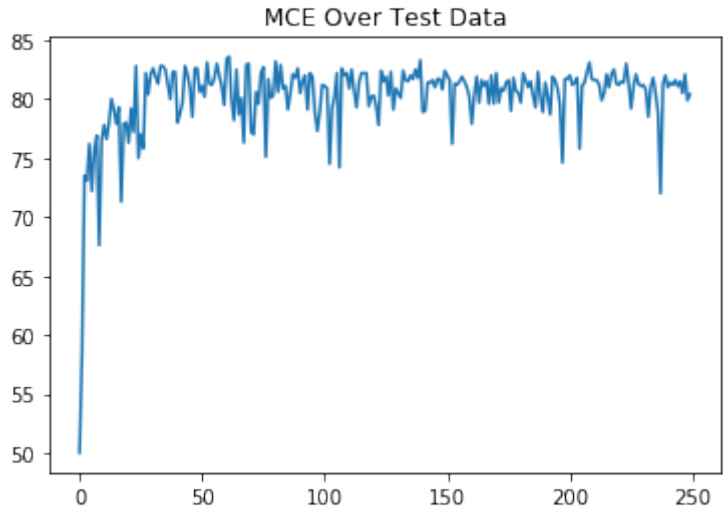
[13]:
```
figureNum += 1
plt.figure(figureNum)
plt.plot(test_mse)
plt.title('MSE Over Test Data')
plt.show()
```



From the figure, we can see that the MSE over test data is converges around 0.6. And after 60-70 epoch, the MSE of the test data starts to increase gradually.

[14]:
```
figureNum += 1
plt.figure(figureNum)
plt.plot(test_mce)
```

```
plt.title('MCE Over Test Data')
plt.show()
```



Also, classification accuracy of the test data reach its maximum around 60-70 epochs and then gradually decrease and converges to 80.4 over 250 epochs. This is because after a point, network starts to memorize the training data and MCE over test data starts to decrease while MSE over test data starts to increase.

[15]:
```
print(test_mce[-1])
```

```
80.4
```

Part b

In this part, we are asked to describe how the squared error and classification error metrics evolve over epochs for the training data versus the testing data. We are also asked whether the squared error an adequate predictor of classification error.

As stated in the part a, for both training and test data, there is a strong correlation between the MSE and MCE since while MSE is decreasing over epochs, MCE increases and vice versa. This holds for both training data and test data. However, MSE might have convergence problems with binary classification. This is because output of a binary classification is passed into the MSE through sigmoid/tanh function. If somehow, the MSE lands on the concave parts of the sigmoid and tanh, then the gradient descent will not work and network will not improve itself or improve itself very slowly.

Even though, there is a correlation of MSE and MCE in the part a, we can say that MSE is good for understanding whether the network is learning, but it is not an adequate error metric for classification.

Part c

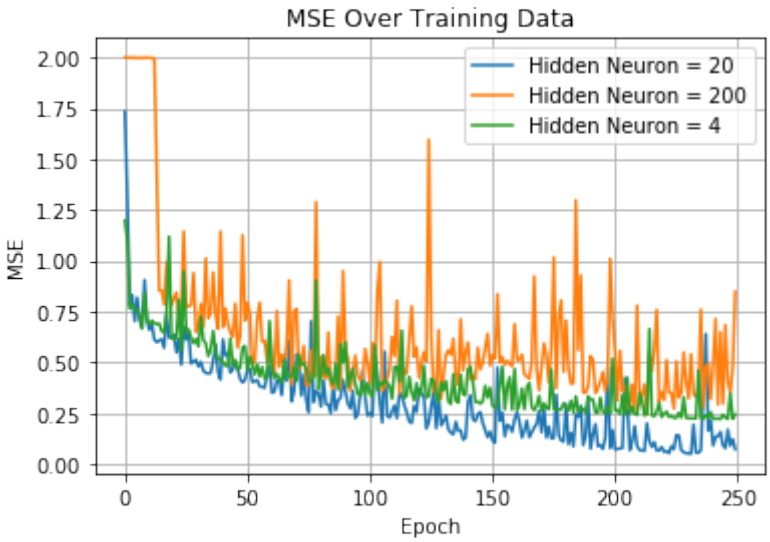
In this part, we are asked to train the network with substantially smaller and substantially higher number of neurons. Here, I have chosen N = 200 for substantialy higher neuron size and N = 4 for substantially smaller neuron size. The training for those networks and the overlay plot is given below:

[16]:
```
# Part C

best_lr = 0.35
high_n = 200

neuralNet = MLP()
```

```
neuralNet.addLayer(HiddenLayer(32**2, high_n, 0, 0.02))
neuralNet.addLayer(HiddenLayer(high_n, 1, 0, 0.02))

mse_loss_highN, mce_loss_highN, test_mse_highN, test_mce_highN = neuralNet.
 ↪trainNetwork(train_img_flat/255, trainlabels, test_img_flat/255, testlabels,␣
 ↪best_lr, 50, 250)
```

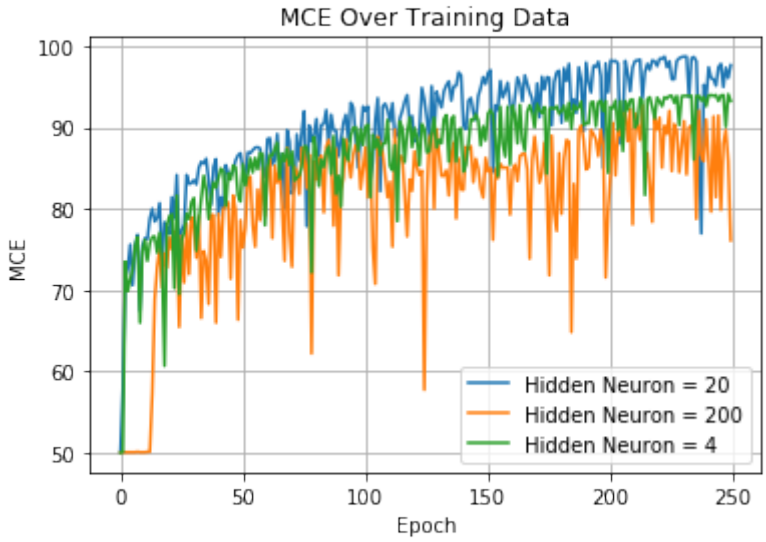
[17]:
```
best_lr = 0.35
low_n = 4

neuralNet = MLP()
neuralNet.addLayer(HiddenLayer(32**2, low_n, 0, 0.02))
neuralNet.addLayer(HiddenLayer(low_n, 1, 0, 0.02))

mse_loss_lowN, mce_loss_lowN, test_mse_lowN, test_mce_lowN = neuralNet.
 ↪trainNetwork(train_img_flat/255, trainlabels, test_img_flat/255, testlabels,␣
 ↪best_lr, 50, 250)
```

[18]:
```
figureNum += 1
plt.figure(figureNum)
plt.plot(mse_loss)
plt.plot(mse_loss_highN)
plt.plot(mse_loss_lowN)

plt.title('MSE Over Training Data')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.legend(["Hidden Neuron = " + str(best_n), "Hidden Neuron = " + str(high_n),␣
 ↪"Hidden Neuron = " + str(low_n)])
plt.grid()
plt.show()
```



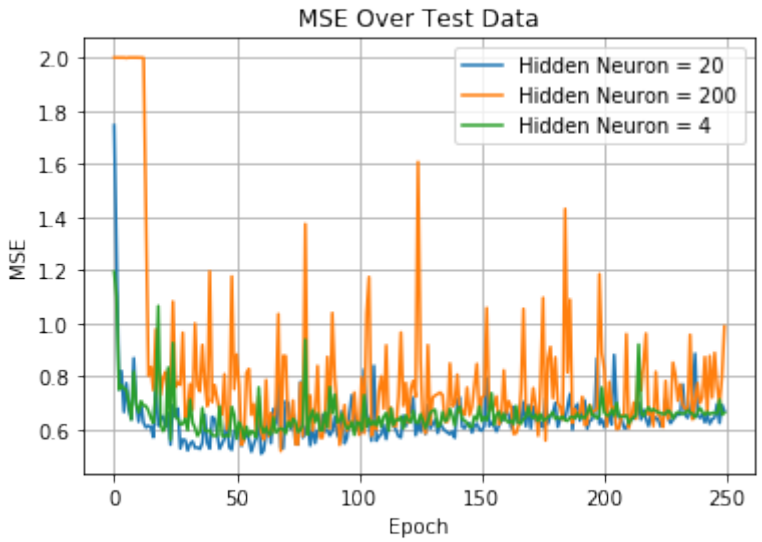
From the figure, we can see that when N=20, MSE converges to 0 over training data and other networks converges to 0.25. Also, networks with substantially higher and substantially lower N, does bigger oscillations while converging.

[19]:
```
figureNum += 1
plt.figure(figureNum)
plt.plot(mce_loss)
plt.plot(mce_loss_highN)
plt.plot(mce_loss_lowN)

plt.title('MCE Over Training Data')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.legend(["Hidden Neuron = " + str(best_n), "Hidden Neuron = " + str(high_n),␣
 ↪"Hidden Neuron = " + str(low_n)])
plt.grid()
plt.show()
```



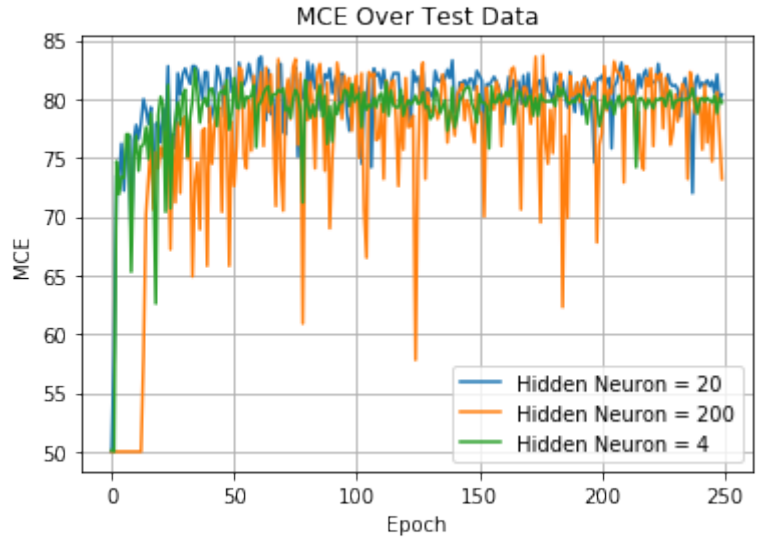
Also, it can be seen that when N=20, classification accuracy is better than the networks with smaller and larger N. Thus, optimal network performs better on the training data.

[20]:
```
figureNum += 1
plt.figure(figureNum)
plt.plot(test_mse)
plt.plot(test_mse_highN)
plt.plot(test_mse_lowN)

plt.title('MSE Over Test Data')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.legend(["Hidden Neuron = " + str(best_n), "Hidden Neuron = " + str(high_n),␣
 ↪"Hidden Neuron = " + str(low_n)])
plt.grid()
plt.show()
```



However,for MSE on the test data, all three networks converges the same. The difference is that optimal network and network with smaller N is more stable and much smaller oscillations than the network with larger neuron size.

[21]:
```
figureNum += 1
plt.figure(figureNum)
plt.plot(test_mce)
plt.plot(test_mce_highN)
plt.plot(test_mce_lowN)

plt.title('MCE Over Test Data')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.legend(["Hidden Neuron = " + str(best_n), "Hidden Neuron = " + str(high_n),␣
 ↪"Hidden Neuron = " + str(low_n)])
plt.grid()
plt.show()
```



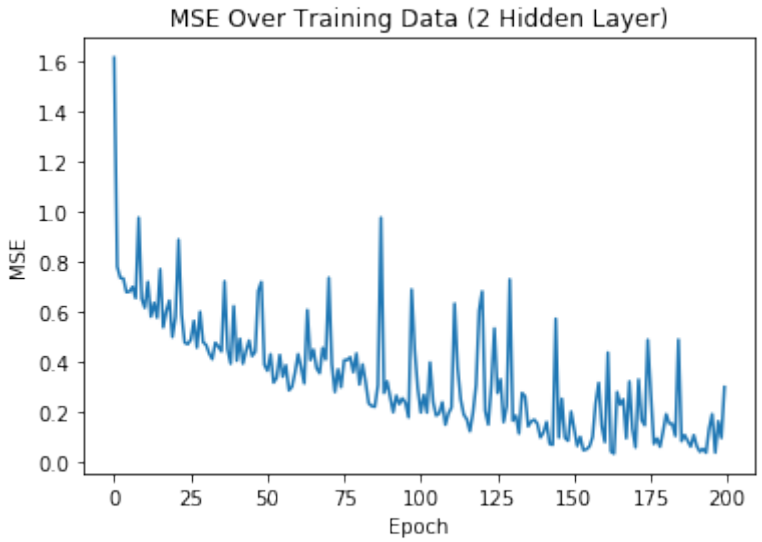
The MCE on the test data, performs a little bit better than the network with smaller neuron size. The network with larger number of hidden neuron still has large oscillations over the test data. However, even though the optimal network performed way better in the training data, the difference is relatively smaller for test data. This may be due to fact that my learning rate was not optimal for the other 2 networks and they could not pass local minimum on training.

Part d

In this part, we are asked to construct and train a seperate network with 2 hidden layers. As parameters, the first layer has 512 neurons, the second layer has 64 neurons, learning rate η is 0.35 and batchsize is 38. This network, MSEs and MCEs on both training and test data is given below:

[22]:
```
# Part D
neuralNet_2L = MLP()
neuralNet_2L.addLayer(HiddenLayer(32**2, 512, 0, 0.02))
neuralNet_2L.addLayer(HiddenLayer(512, 64, 0, 0.02))
neuralNet_2L.addLayer(HiddenLayer(64, 1, 0, 0.02))

mse_loss_2l, mce_loss_2l, test_mse_2l, test_mce_2l = neuralNet_2L.
↪trainNetwork(train_img_flat/255, trainlabels,test_img_flat/255, testlabels, 0.35,␣
↪50, 200)
```

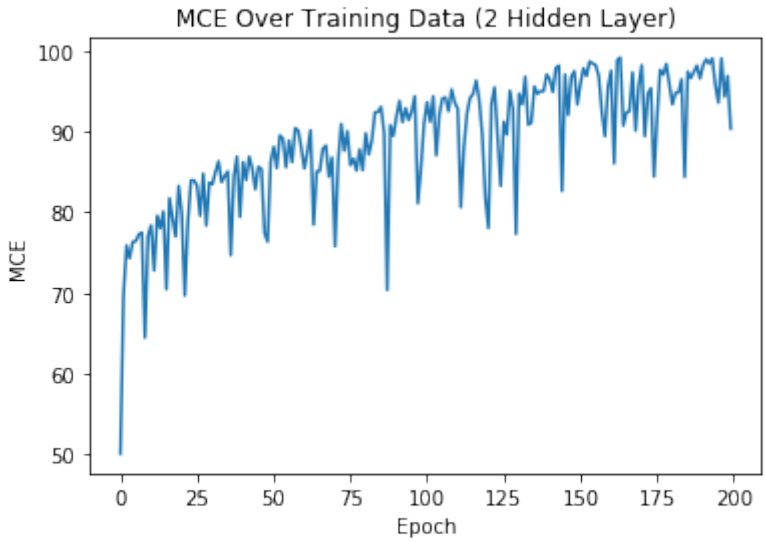
[23]:
```
figureNum += 1
plt.figure(figureNum)
plt.title('MSE Over Training Data (2 Hidden Layer)')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.plot(mse_loss_2l)
plt.show()
```



The MSE over training data is given above. It can be seen that the network with 2 layers has more oscillatory behaviour than the network in part a. The network with 1 hidden layer converges in a much more stable way.

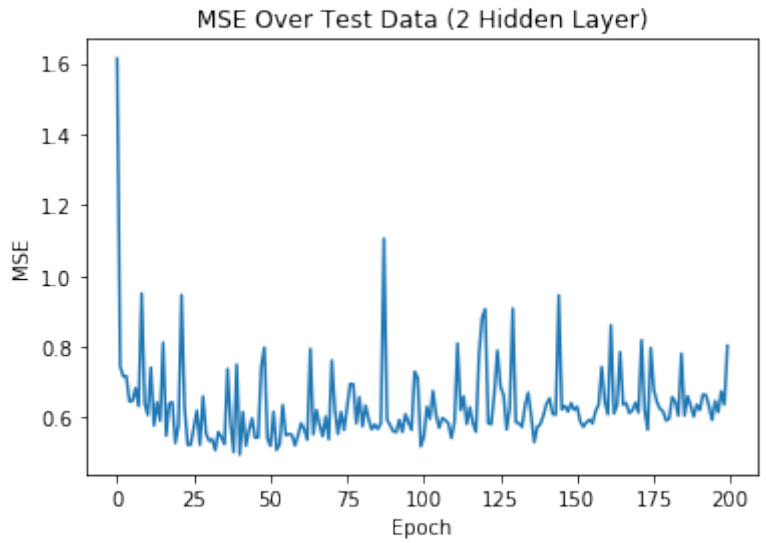
[24]:
```
figureNum += 1
plt.figure(figureNum)
```

```
plt.title('MCE Over Training Data (2 Hidden Layer)')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.plot(mce_loss_2l)
plt.show()
```



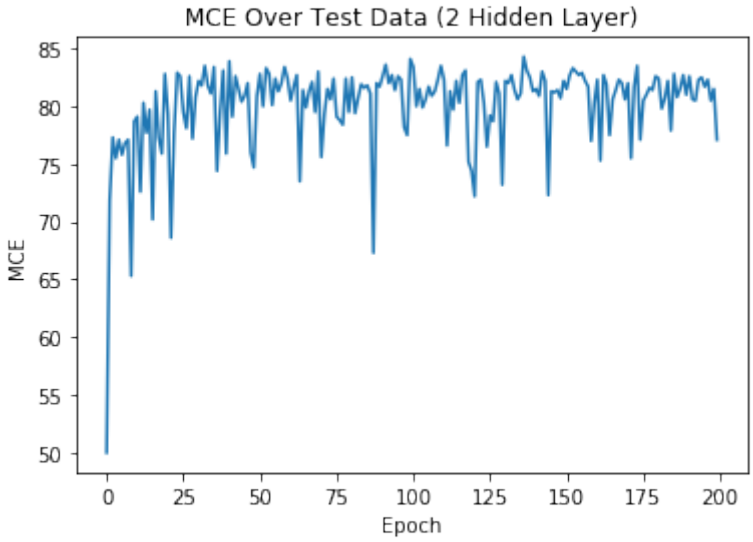
The MCE over training data is given above. The network with 1 hidden layer converges to 100% accuracy on the training data, while the network with 2 hidden layer does more oscillations and less stable than the other network. This may be due to the network with 2 hidden layer is more complex than the complexity of the data. So, the network with 1 hidden layer performed better on the training data.

[25]:
```
figureNum += 1
plt.figure(figureNum)
plt.title('MSE Over Test Data (2 Hidden Layer)')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.plot(test_mse_2l)
plt.show()
```



The MSE over test data is given above. Here, it can be seen that the convergence is faster than the network with 1 hidden layer. This is expected since the network is deeper and more complex, thus the learning is faster. However, we also can see that the network with 2 hidden layer again is less stable since it shows more oscillatory behaviour.

[26]:
```
figureNum += 1
plt.figure(figureNum)
plt.title('MCE Over Test Data (2 Hidden Layer)')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.plot(test_mce_2l)
plt.show()
```



The MCE over test data is given above. It can be seen that the convergence is faster again than the network with 1 hidden layer. Since, the network is deeper the convergence is faster. However, again the network with 2 hidden layer performs a more oscillatory behaviour.

Part e

In this part, the same network with 2 hidden layer is trained. However, for the update of weights, momentum is used. Momentum update is used for update the weights with the moving average of the previous weights updates so that the convergence becomes more stable. We can express the momentum update as:

$$\Delta W_{ji}(n) = n \sum_{i=0}^{n} \beta^{n-i} \frac{\partial O(i)}{\partial W_{ji}(i)}$$

where, β is the momentum coefficient and j represents the weights of j^th layer. Thus, the weight updates becomes:

$$W_j(n) = W_j(n) - \eta \frac{\partial O(n)}{\partial W_j(n)} - \beta \Delta W_j(n-1)$$

The python implementation can be shown as:

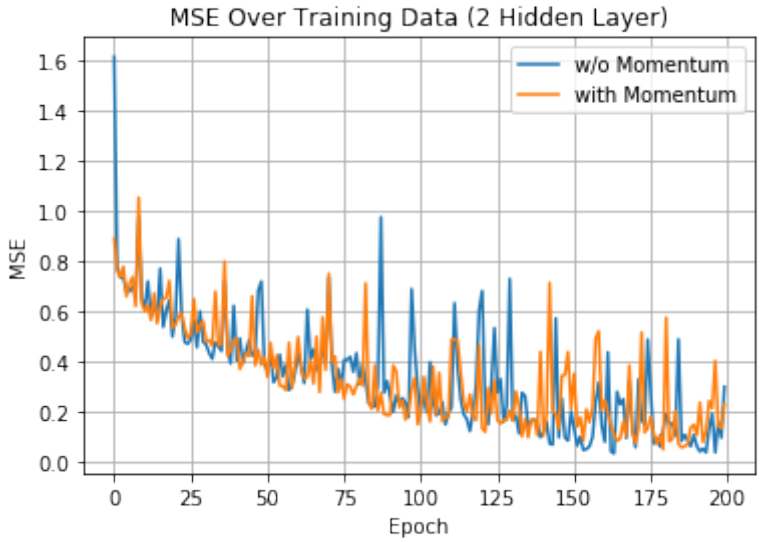
[27]:
```
def update_weights(self, data, label, learningRate):
        '''
        update_weights function updates the weights with the gradients found with␣
↪back_propagation.
        INPUTS:

            data          : data is the input.
            label         : label is the labels of the data.
            learnigRate   : learningRate is the coefficient for the weight update.

        RETURNS:

        '''


        layerSize = np.shape(self.layers)[0]
        self.back_propagation(data,label)
        # If momentum is used.
        if( self.momentum == True ):
            for i in range(layerSize):

                self.layers[i].momentum_dw = self.layers[i].dW + (self.momentumCoef␣
↪* self.layers[i].momentum_dw)
                self.layers[i].momentum_db = self.layers[i].dB + (self.momentumCoef␣
↪* self.layers[i].momentum_db)

                self.layers[i].weights -= (learningRate * self.layers[i].
↪momentum_dw)/self.batchSize
                self.layers[i].bias -= (learningRate * self.layers[i].momentum_db)/
↪self.batchSize

        #If momentum is not used.
        else:
            for i in range(layerSize):

                self.layers[i].weights -= (learningRate * self.layers[i].dW)/self.
↪batchSize
                self.layers[i].bias -= (learningRate * self.layers[i].dB)/self.
↪batchSize
```

The same network with 2 hidden layer is trained with momentum updates where the momentum coefficient is the 0.2. The comparisons of the MSEs and MCEs on both training and test data is given below for the network with momentum and without momentum is given below.

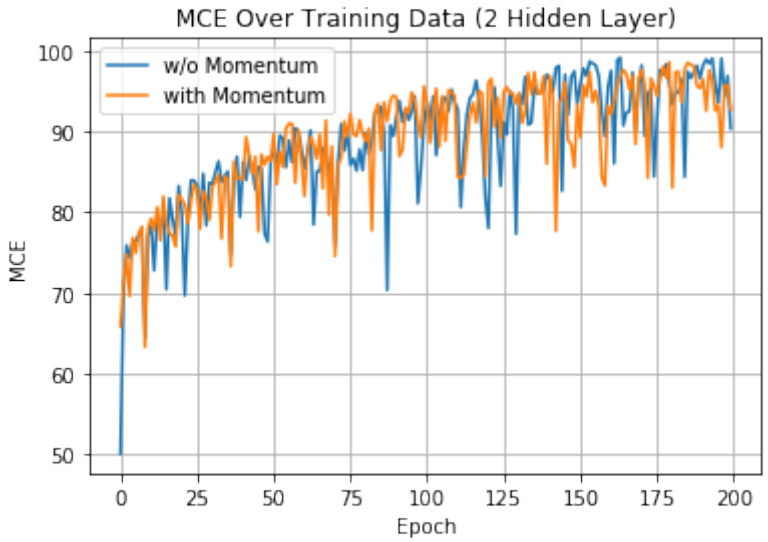
[27]:
```
# Part E

neuralNet_2L = MLP(momentum = True, momentumCoef = 0.2)
neuralNet_2L.addLayer(HiddenLayer(32**2, 512, 0, 0.02))
neuralNet_2L.addLayer(HiddenLayer(512, 64, 0, 0.02))
neuralNet_2L.addLayer(HiddenLayer(64, 1, 0, 0.02))

mse_loss_2l_m, mce_loss_2l_m, test_mse_2l_m, test_mce_2l_m = neuralNet_2L.
↪trainNetwork(train_img_flat/255, trainlabels,test_img_flat/255, testlabels, 0.35,␣
↪50, 200)
```

[29]:
```
figureNum += 1
plt.figure(figureNum)
```

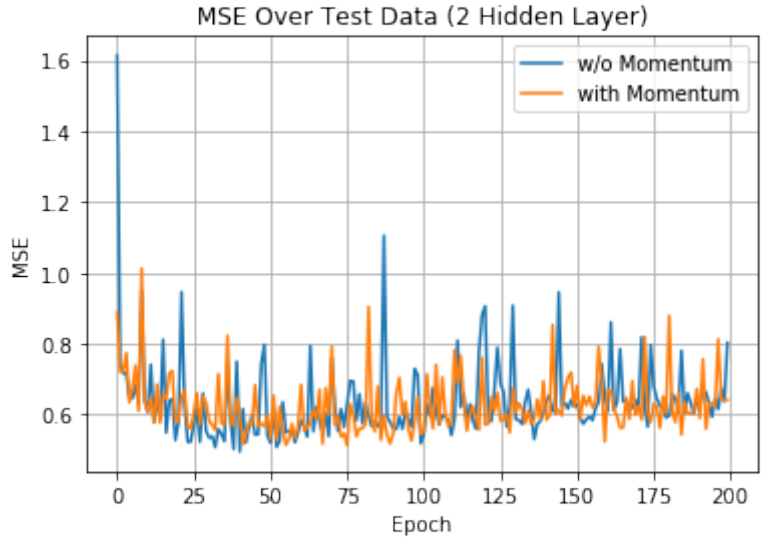
```
plt.plot(mse_loss_2l)
plt.plot(mse_loss_2l_m)

plt.title('MSE Over Training Data (2 Hidden Layer)')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.legend(["w/o Momentum", "with Momentum"])
plt.grid()
plt.show()
```



[30]:
```
figureNum += 1
plt.figure(figureNum)
plt.plot(mce_loss_2l)
plt.plot(mce_loss_2l_m)

plt.title('MCE Over Training Data (2 Hidden Layer)')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.legend(["w/o Momentum", "with Momentum"])
plt.grid()
plt.show()
```

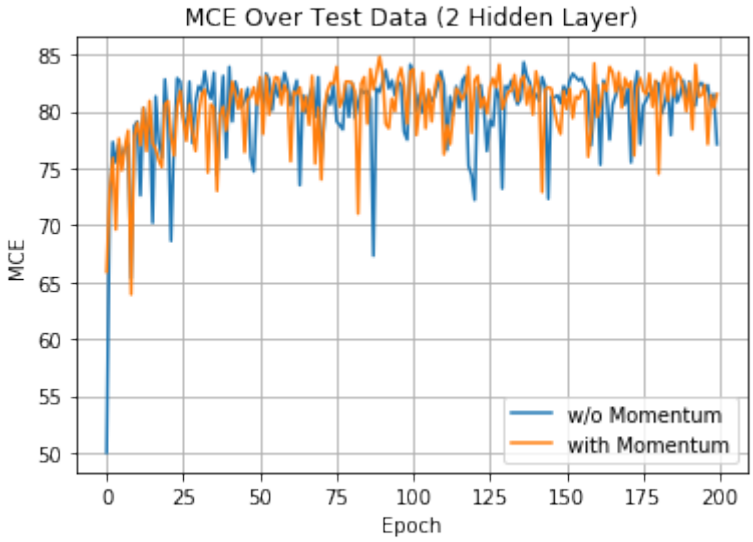


[31]:
```
figureNum += 1
plt.figure(figureNum)
plt.plot(test_mse_2l)
plt.plot(test_mse_2l_m)

plt.title('MSE Over Test Data (2 Hidden Layer)')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.legend(["w/o Momentum", "with Momentum"])
plt.grid()
plt.show()
```



[32]:
```
figureNum += 1
plt.figure(figureNum)
plt.plot(test_mce_2l)
plt.plot(test_mce_2l_m)
```

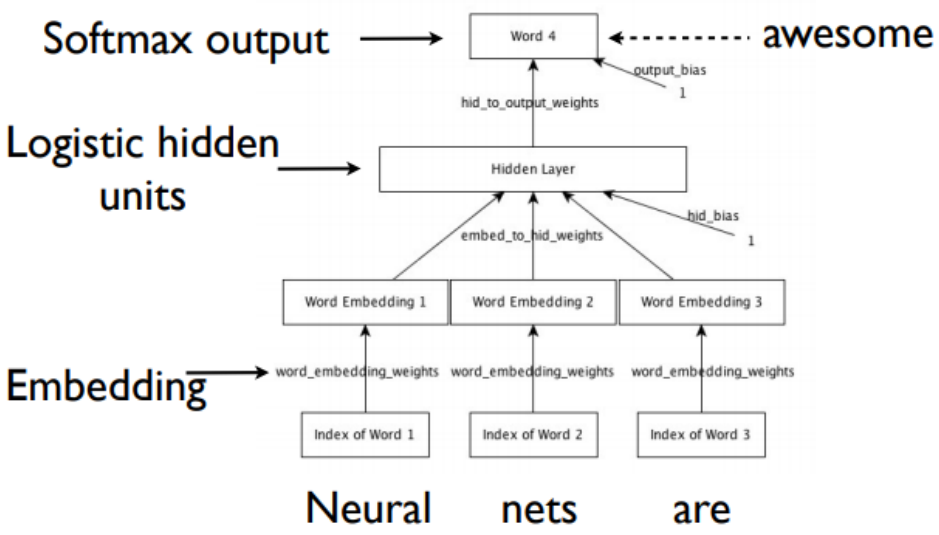
```
plt.title('MCE Over Test Data (2 Hidden Layer)')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.legend(["w/o Momentum", "with Momentum"])
plt.grid()
plt.show()
```



From the figures, we can observe that the momentum update made the network more stable. Network does not perform better but the oscillatory behaviour of the network is much more stabilized which is what we have expected with the momentum update.

Question 2

In this question, we are going to implement a network which given a triagram, predicts the 4^th word. The words matrix gives us the unique words which has size of 250. The network structure looks like:



The input layer has 3 neurons which are the entries for the triagram. An embedding layer is used to linearly map the words into a vector representation of length D. Transforming the input to one-hot encoded inputs, the embedding matrix R has the size of (250,D). This embedding matrix is used for each word in the triagram without considering the sequence of the words. Moreover, the hidden layer usu a sigmoid activation function and the output uses softmax.

Part a

We are given that the network should train with a batch size of 200, a learning rate $\eta = 0.15$, a momentum coefficient of 0.85. The weights and bias should be initialized from a Gaussian Distribution with 0.01 standard deviation. As error metric, cross entropy is used.

First, the transformation of the data to one-hot encoded data should be done. Since, there are 250 words, each word should be transformed into a matrix size of 250 where the only the word's corresponding index is 1 and 0 otherwise. So, the data transformation of the data into one-hot encoded data is given below.

[1]:
```
import numpy as np
import h5py
import matplotlib.pyplot as plt
```

[2]:
```
# Part A

f = h5py.File('assign2_data2.h5', 'r')
dataKeys = list(f.keys())
print('The data keys are:' + str(dataKeys))

# Gathering the  train images, test images, train labels and test labels.
testdata = np.asarray(f['testx'])
testlbls = np.asarray(f['testd'])
traindata = np.asarray(f['trainx'])
trainlbls = np.asarray(f['traind'])
validdata = np.asarray(f['valx'])
validlbls = np.asarray(f['vald'])
```

```
words = np.asarray(f['words'])

print('The size of testdata is: ' + str(np.shape(testdata)))
print('The size of testlbls is: ' + str(np.shape(testlbls)))
print('The size of traindata is: ' + str(np.shape(traindata)))
print('The size of trainlbls is: ' + str(np.shape(trainlbls)))
print('The size of validdata is: ' + str(np.shape(validdata)))
print('The size of validlbls is: ' + str(np.shape(validlbls)))
print('The size of words is: ' + str(np.shape(words)))
```

```
The data keys are:['testd', 'testx', 'traind', 'trainx', 'vald', 'valx',
'words']
The size of testdata is: (46500, 3)
The size of testlbls is: (46500,)
The size of traindata is: (372500, 3)
The size of trainlbls is: (372500,)
The size of validdata is: (46500, 3)
The size of validlbls is: (46500,)
The size of words is: (250,)
```

[3]:
```
# One Hot Encode Transform

def one_hot_encoder(x):
    samplesize = np.shape(x)[0]
    x = x.reshape(samplesize,-1)
    featuresize = np.shape(x)[1]
    result = np.zeros((samplesize, featuresize, 250))
    for i in range(samplesize):
        for j in range(featuresize):
            a = x[i,j]
            result[i,j,a-1] = 1

    if (featuresize == 1):
        result = result.reshape(-1,250)
    else:
        result = result.reshape(samplesize,250,featuresize)
    return result
```

[4]:
```
train_data = one_hot_encoder(traindata)
train_labels = one_hot_encoder(trainlbls)
test_data = one_hot_encoder(testdata)
test_labels = one_hot_encoder(testlbls)
val_data = one_hot_encoder(validdata)
val_labels = one_hot_encoder(validlbls)

print('The size of test_data is: ' + str(np.shape(test_data)))
print('The size of test_labels is: ' + str(np.shape(test_labels)))
print('The size of train_data is: ' + str(np.shape(train_data)))
print('The size of train_labels is: ' + str(np.shape(train_labels)))
print('The size of val_data is: ' + str(np.shape(val_data)))
print('The size of val_labels is: ' + str(np.shape(val_labels)))
```

```
The size of test_data is: (46500, 250, 3)
The size of test_labels is: (46500, 250)
The size of train_data is: (372500, 250, 3)
The size of train_labels is: (372500, 250)
The size of val_data is: (46500, 250, 3)
```

```
The size of val_labels is: (46500, 250)
```

Here, the data transformation into one-hot encoded data is completed. Then, the HiddenLayer and MLP class from the Question1 is used with slight changes to activation function, loss function and an additional embedding layer. The activation function of the hidden layer is sigmoid which can be written as:

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

and the derivative of the sigmoid function der_sigmoid can be written as:

$$\frac{\partial \sigma(x)}{\partial x} = der\_sigmoid = \sigma(x)(1-\sigma(x))$$

Then, the soft-max which will be used in the output layer can be expressed as:

$$softmax(z_i) = o_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

and the derivative of the soft-max function der_softmax can be written as:

$$\frac{\partial o_i}{\partial z_i} = der\_softmax(z_i) == softmax(z_i)(1 - softmax(z_i))$$

The error metric of the network cross-entropy can be written as:

$$cross\_entropy(y, \hat{y}) = -\frac{1}{n}\sum_{j=1}^{n} y_j * log(\hat{y}_j) + (1-y_j) * (1 - log(\hat{y}_j))$$

and the derivative of the cross entropy der_cross_entropy is as:

$$der\_cross\_entropy(y, \hat{y}) = -(y/\hat{y}) + (1-y)/(1-\hat{y})$$

Finally, for the embedding layer, the activation function(we) can be written as:

$$we(x) = x$$

What that means is that with the weights from the embedding layer, it only used for mapping the words into a vector representation of length D (250,D). Therefore the derivative of we which is der_we(x) is:

$$\frac{\partial we(x)}{\partial x} = der\_we(x) = 1$$

Thus, the der_we(x) will return a matrix of 1's with the same shape of input x. The python code for updated HiddenLayer and MLP class is given below:

[5]:
```
class HiddenLayer:
    def __init__(self, neuronNum, neuronSize, mean, std):

        '''
        This class creates a hidden laer for neural network.
        Weights and bias are initially random Gaussian distribution.

        INPUTS:

            neuronNum    : neuronNum is the features a neuron holds.
            neuronSize   : neuronSize is the number of neurons in a hidden layer.
            mean         : mean for Gaussian distribution.
            std          : Standard deviation for Gaussian distribution.
```

```
        RETURNS:

        '''

        np.random.seed(8)
        self.weights = np.random.normal( loc=mean, scale=std,␣
↪size=(neuronNum,neuronSize))
        self.bias = np.random.normal( loc=mean, scale=std, size=(1,neuronSize))
        self.Z = None
        self.A = None
        self.grad = None
        self.dB = None
        self.dW = None
        self.error = None
        self.momentum_dw = 0
        self.momentum_db = 0
```

[6]:
```
class MLP:
    def __init__(self, momentum = False, momentumCoef = 0):

        '''
        This class creates a multilayer perceptron network.

        INPUTS:

            momentum     : momentum is the boolean for the network which indicates
                           whether the momentum learning will be done or not

            momentumCoef : Coefficient of momentum learning


        RETURNS:

        '''

        self.momentum = momentum
        self.momentumCoef = momentumCoef
        self.layers = list()
        self.batchSize = 0


    def addLayer(self, layer):

        '''
        This function adds a HiddenLayer class to the network.

        INPUTS:

            layer        : layer is an instance of HiddenLayer class

        RETURNS:

        '''

        self.layers.append(layer)

    def sigmoid(self, x):
```

```
        '''
        This function is the sigmoid for the activation function.
        INPUTS:

            x            : x is the weighted sum which will be pushed to activation␣
↪function.

        RETURNS:

            result       : result is the sigmoid of the input x.
        '''

        result = 1 / (1 + np.exp(-x))
        return result

    def der_sigmoid(self, x):
        '''
        This function is the derivative of sigmoid function.
        INPUTS:

            x            : x is the input.

        RETURNS:

            result       : result is the derivative of sigmoid of the input x.
        '''


        result = self.sigmoid(x) * (1 - self.sigmoid(x))
        return result

    def softmax(self, x):

        '''
        This function is the softmax for the activation function of output layer.
        INPUTS:

            x            : x is the weighted sum which will be pushed to activation␣
↪function.

        RETURNS:

            result       : result is the softmax of the input x.
        '''

        e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
        result = e_x / np.sum(e_x, axis=-1, keepdims=True)
        return result

    def der_softmax(self, x):

        '''
        This function is the derivative of softmax.
        INPUTS:

            x            : x is the input.
```

```
        RETURNS:

            result        : result is the derivative of softmax of the input x.
        '''

        p = self.softmax(x)
        result = p * (1-p)
        return result

    def we(self, x):

        '''
        This function is the word embedding layer's activation function.
        INPUTS:

            x             : x is the weighted sum which will be pushed to activation␣
↪function.

        RETURNS:

            result        : result is the x itself.
        '''

        return x

    def der_we(self, x):
        '''
        This function is the derivative of word embedding layer's activation␣
↪function.
        INPUTS:

            x             : x is input.

        RETURNS:

            result        : result is an array of ones with x's shape..
        '''


        return np.ones(x.shape)

    def cross_entropy(self, y, y_pred):

        '''
        cross_entropy is the loss function for the network.
        INPUTS:

            y             : y is the labels for our data.
            y_pred        : y_pred is the network's prediction.

        RETURNS:

            cost          : cost is the cross entropy error between y and y_pred.
        '''

        # To avoid 0
```

```
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)

        cost = -np.mean(y*np.log(y_pred) + (1-y)*np.log(1-y_pred))
        return cost

    def der_cross_entropy(self, y, y_pred):
        '''
        der_cross_entropy is the derivative of loss function for the network.
        INPUTS:

            y             : y is the labels for our data.
            y_pred        : y_pred is the network's prediction.

        RETURNS:

                          : returns the derivative of cross entropy error between y␣
↪and y_pred.
        '''

        # To avoid division by 0.
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)

        return - (y / y_pred) + (1 - y) / (1 - y_pred)

    def forward(self, data):

        '''
        forward function is the forward propagation.
        INPUTS:

            data          : data is the input which will pushed to forward␣
↪propagation.

        RETURNS:

                          : returns the prediction of the network.
        '''

        layerSize = np.shape(self.layers)[0]
        for i in range(layerSize):
            # Output Layer
            if (i == layerSize - 1):
                self.layers[i].A = (self.layers[i-1].Z).dot(self.layers[i].weights)␣
↪+ self.layers[i].bias
                self.layers[i].Z = self.softmax(self.layers[i].A)
            # 1st Hidden Layer
            if (i == 0):
                self.layers[i].A = data.dot(self.layers[i].weights) + self.
↪layers[i].bias
                self.layers[i].Z = self.we(self.layers[i].A)
            # Other Hidden Layers
            else:
                self.layers[i].A = (self.layers[i-1].Z).dot(self.layers[i].weights)␣
↪+ self.layers[i].bias
                self.layers[i].Z = self.sigmoid(self.layers[i].A)

        return self.layers[-1].Z
```

```
    def back_propagation(self, data, label):

        '''
        back_propagation function is the back propagation algorithm for weight and␣
↪bias updates.
        back_propagation function first calls forward function to predict the␣
↪output of network which us y_pred.
        INPUTS:

            data          : data is the input.
            label         : label is the labels of the data.

        RETURNS:

        '''

        layerSize = np.shape(self.layers)[0]
        y_pred = self.forward(data)

        for i in range(layerSize)[::-1]:
            # Output Layer
            if (i == layerSize - 1):
                self.layers[i].error = self.der_cross_entropy(label, y_pred)
                self.layers[i].grad = (self.layers[i].error) * (self.
↪der_softmax(self.layers[i].A))
                self.layers[i].dW = (self.layers[i-1].Z).T.dot(self.layers[i].grad)
                self.layers[i].dB = np.sum(self.layers[i].grad, axis=0,␣
↪keepdims=True)

            # 1st Hidden Layer
            elif(i == 0):
                self.layers[i].error = (self.layers[i+1].grad).dot(self.layers[i+1].
↪weights.T)
                self.layers[i].grad = (self.layers[i].error) * self.der_we(self.
↪layers[i].A)
                self.layers[i].dW = data.T.dot(self.layers[i].grad)
                self.layers[i].dB = np.sum(self.layers[i].grad, axis=0,␣
↪keepdims=True)

            # Other Layers
            else:
                self.layers[i].error = (self.layers[i+1].grad).dot(self.layers[i+1].
↪weights.T)

                self.layers[i].grad = (self.layers[i].error) * self.
↪der_sigmoid(self.layers[i].A)
                self.layers[i].dW = (self.layers[i-1].Z).T.dot(self.layers[i].grad)
                self.layers[i].dB = np.sum(self.layers[i].grad, axis=0,␣
↪keepdims=True)


    def update_weights(self, data, label, learningRate):

        '''
```

```
        update_weights function updates the weights with the gradients found with␣
↪back_propagation.
        INPUTS:

            data          : data is the input.
            label         : label is the labels of the data.
            learnigRate   : learningRate is the coefficient for the weight update.

        RETURNS:

        '''

        layerSize = np.shape(self.layers)[0]
        self.back_propagation(data,label)

        # If momentum is used.
        if( self.momentum == True ):
            for i in range(layerSize):

                self.layers[i].momentum_dw = self.layers[i].dW + (self.momentumCoef␣
↪* self.layers[i].momentum_dw)
                self.layers[i].momentum_db = self.layers[i].dB + (self.momentumCoef␣
↪* self.layers[i].momentum_db)

                self.layers[i].weights -= (learningRate * self.layers[i].
↪momentum_dw)/self.batchSize
                self.layers[i].bias -= (learningRate * self.layers[i].momentum_db)/
↪self.batchSize

        # If momentum is not used.
        else:
            for i in range(layerSize):

                self.layers[i].weights -= (learningRate * self.layers[i].dW)/self.
↪batchSize
                self.layers[i].bias -= (learningRate * self.layers[i].dB)/self.
↪batchSize


    def predict(self, y_pred):

        '''
        predict function predicts and output from the network's output y_pred.
        INPUTS:

            y_pred        : MLP's output.

        RETURNS:

                          : returns the label for prediction of the network.
        '''

        word = np.argmax(y_pred, axis=-1)
        return word

    def accuracy(self, y, y_pred):
```

```
        '''
        accuracy function is the accuracy of our network.
        INPUTS:

            y            : y is the labels for our data.
            y_pred       : y_pred is the network's prediction.

        RETURNS:

                         : returns the accuracy between y and y_pred.
        '''

        count = 0

        y_words = np.argmax(y, axis=-1)
        y_pred_words = self.predict(y_pred)
        for i in range(len(y_words)):
            if(y_words[i] == y_pred_words[i]):
                count += 1
        return 100 * (count / len(y_words))

    def predict_10(self, y, y_pred,k,seed):

        '''
        predict_10 function lists the top 10 candidates for the selected triagram.
        INPUTS:

            y            : y is the labels for our data.
            y_pred       : y_pred is the network's prediction.
            k            : K is the number of predictions.
            seed         : For the random initialization.

        RETURNS:

            indeces           : indeces is the indeces of the selected 5 samples.
            indeces_test      : indeces_test is the labels of the selected 5␣
↪sampled from test data.
            indeces_test_pred : indeces_test_pred is the prediction of the network␣
↪for the selected 5 samples
                                from the test data.
        '''

        np.random.seed(seed)
        indeces = np.random.randint(250, size=5)
        y_new = y[indeces]
        y_pred_new = y_pred[indeces]
        indeces_test_pred = list()
        indeces_test = list()

        for i in range(np.shape(y_pred_new)[0]):
            indeces_test.append(np.argmax(y[i], axis=-1))
            indeces_test_pred.append(np.argpartition(y_pred_new[i], -k, axis=-1)[-k:
↪])

        return indeces, indeces_test, indeces_test_pred
```

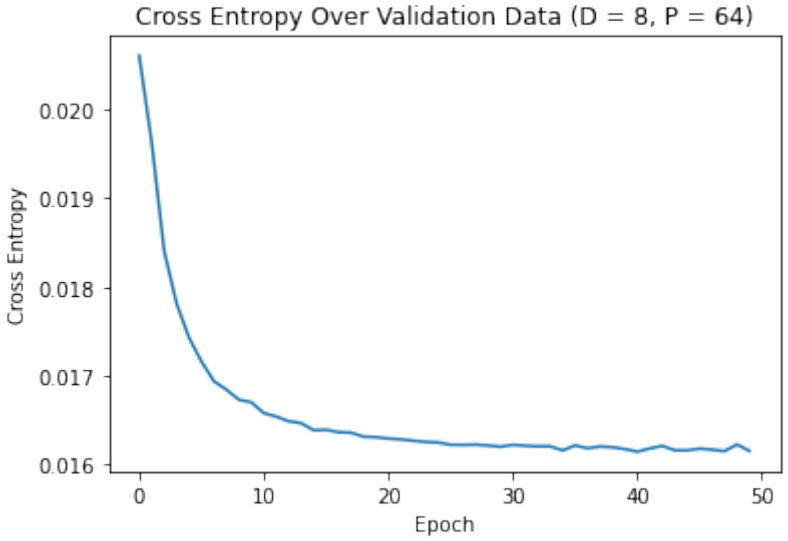
```
   def trainNetwork(self, data, label, validData, validLabel, learningRate,␣
↪batchNum, epoch):

        '''
        trainNetwork function calls the update_weights function to train the␣
↪network over mini-batches
        for fiven number of epochs.
        INPUTS:

            data          : data is the training data.
            label         : label is the labels of the data.
            validData     : validData is the valid data.
            validLabel    : validLabel is the labels of the validData.
            learningRate  : learningRate is the coefficient for the weight update.
            batchNum      : batchNum is the number of mini-batches.
            epoch         : Number of times the network train the whole data.

        RETURNS:

            cost          : Cross entropy error over epochs on test data.
            accuracies    : Accuracies over epochs on test data.

        '''

        cost = list()
        accuracies = list()

        np.random.seed(7)

        for i in range(epoch):
            randomIndexes = np.random.permutation(len(label))
            data = data[randomIndexes]
            label = label[randomIndexes]

            batchLength = len(label) / batchNum
            self.batchSize = batchLength
            for j in range(batchNum):

                start = int(batchLength*j)
                end = int(batchLength*(j+1))
                self.update_weights(data[start:end],label[start:end],learningRate)

            y_pred = self.forward(validData)
            cross_entropy_cost = self.cross_entropy(validLabel, y_pred)
            cost.append(cross_entropy_cost)


            accuracy = self.accuracy(validLabel, y_pred)
            accuracies.append(accuracy)


        return cost, accuracies
```

Here, the network will be trained with different D and P values for (D,P) = (8, 64), (18,128) and (32, 256). Before training the network, data manipulation to training, test and validation data should be done. It is given in the question that the sequence of the words should not be considered. Thus, the training data with size (372500, 250, 3) should be summed over the last axis to assemble a matrix of (372500, 250). The difference between the two matrix is that the each sample in the second matrix has three 1's in the

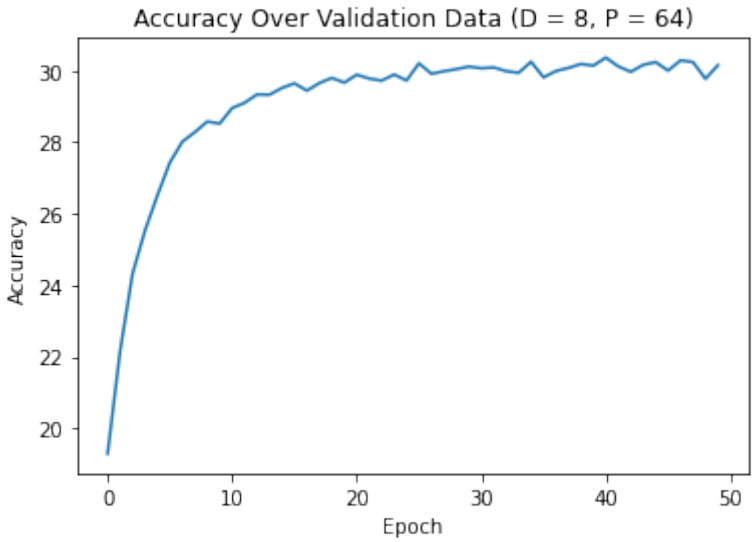
250 one-hot encoded vector. The reason is that we have mapped the 3 words with 250 size into a vector with 250 size which have 1's where the indexes of those 3 words should be. This mapping does not take into consideration of the sequence of the words. However, since it is given that the sequence of the words should not be considered, this mapping is viable. So, the mapping is done to training, test and validation data. Then, for each of the D,P values, the Cross Entropy over validation data and accuracy over validation data is plotted below:

[7]:
```
D = 8
P = 64

neuralNet0 = MLP(momentum = True, momentumCoef = 0.85)
neuralNet0.addLayer(HiddenLayer(250, D, 0, 0.01))
neuralNet0.addLayer(HiddenLayer(D, P, 0, 0.01))
neuralNet0.addLayer(HiddenLayer(P, 250, 0, 0.01))

train_data_1H = np.sum(train_data, axis=-1)
val_data_1H = np.sum(val_data, axis=-1)

cost0, accuracy0 = neuralNet0.trainNetwork(train_data_1H, train_labels,␣
↪val_data_1H, val_labels, 0.15, 1490, 50)
```

[8]:
```
figureNum = 0
plt.figure(figureNum)
plt.title('Cross Entropy Over Validation Data (D = 8, P = 64)')
plt.xlabel('Epoch')
plt.ylabel('Cross Entropy')
plt.plot(cost0)
plt.show()
```

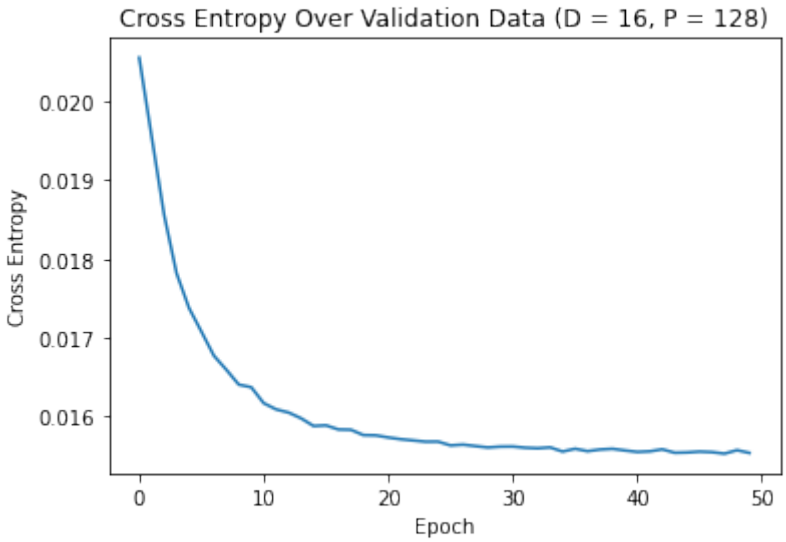


[9]:
```
figureNum += 1
plt.figure(figureNum)
plt.title('Accuracy Over Validation Data (D = 8, P = 64)')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.plot(accuracy0)
plt.show()
```

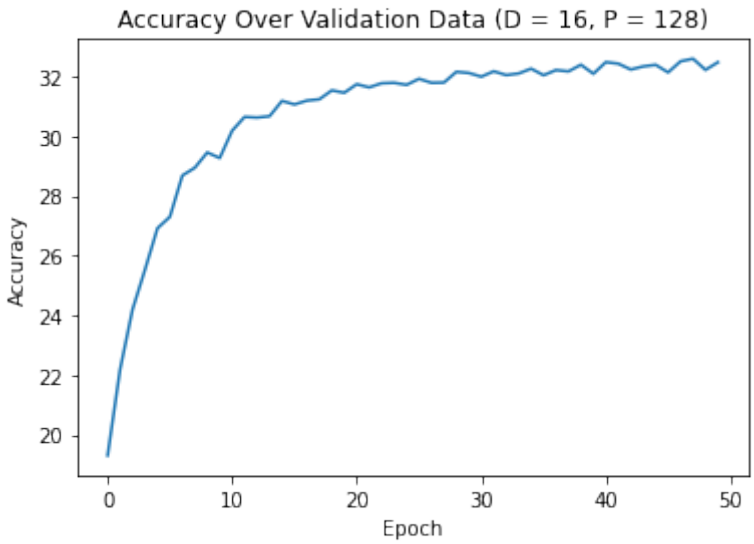


[10]:
```
D = 16
P = 128

neuralNet1 = MLP(momentum = True, momentumCoef = 0.85)
neuralNet1.addLayer(HiddenLayer(250, D, 0, 0.01))
neuralNet1.addLayer(HiddenLayer(D, P, 0, 0.01))
neuralNet1.addLayer(HiddenLayer(P, 250, 0, 0.01))

train_data_1H = np.sum(train_data, axis=-1)
val_data_1H = np.sum(val_data, axis=-1)

cost1, accuracy1 = neuralNet1.trainNetwork(train_data_1H, train_labels,␣
 ↪val_data_1H, val_labels, 0.15, 1490, 50)
```

[11]:
```
figureNum += 1
plt.figure(figureNum)
plt.title('Cross Entropy Over Validation Data (D = 16, P = 128)')
plt.xlabel('Epoch')
plt.ylabel('Cross Entropy')
plt.plot(cost1)
plt.show()
```



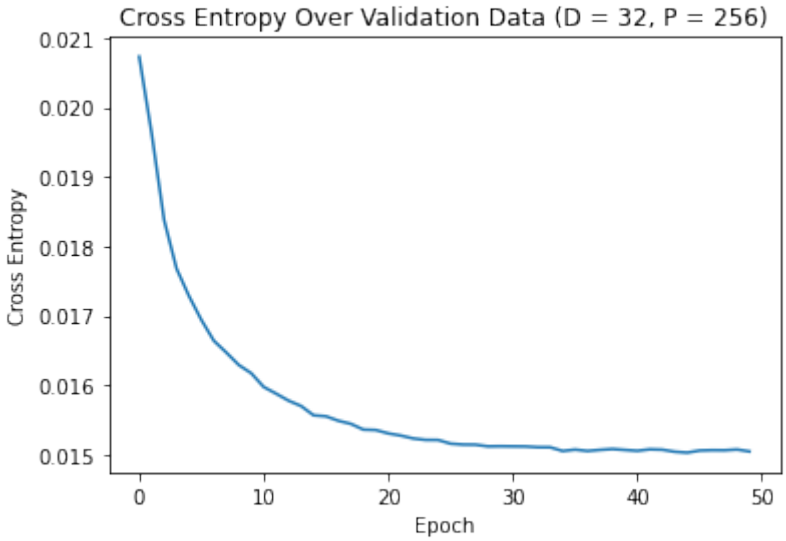
[12]:
```
figureNum += 1
plt.figure(figureNum)
plt.title('Accuracy Over Validation Data (D = 16, P = 128)')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.plot(accuracy1)
plt.show()
```



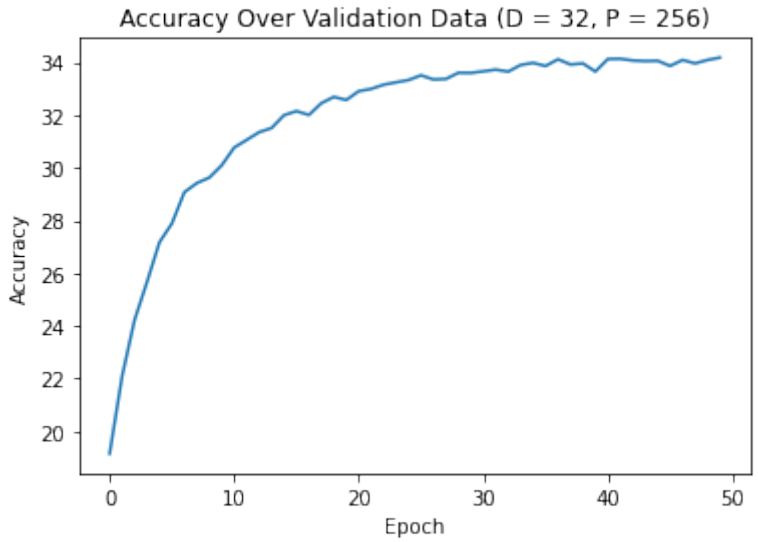
[13]:
```
D = 32
P = 256

neuralNet2 = MLP(momentum = True, momentumCoef = 0.85)
neuralNet2.addLayer(HiddenLayer(250, D, 0, 0.01))
neuralNet2.addLayer(HiddenLayer(D, P, 0, 0.01))
neuralNet2.addLayer(HiddenLayer(P, 250, 0, 0.01))
```

```
train_data_1H = np.sum(train_data, axis=-1)
val_data_1H = np.sum(val_data, axis=-1)

cost2, accuracy2 = neuralNet2.trainNetwork(train_data_1H, train_labels,␣
 ↪val_data_1H, val_labels, 0.15, 1490, 50)
```

[14]:
```
figureNum += 1
plt.figure(figureNum)
plt.title('Cross Entropy Over Validation Data (D = 32, P = 256)')
plt.xlabel('Epoch')
plt.ylabel('Cross Entropy')
plt.plot(cost2)
plt.show()
```



[15]:
```
figureNum += 1
plt.figure(figureNum)
plt.title('Accuracy Over Validation Data (D = 32, P = 256)')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.plot(accuracy2)
plt.show()
```



It can be seen from the graphs, that when (D,P) values go from (8,64) to (16, 128) to (32, 256), the cross entropy loss converging to lower values and the accuracy is slightly better. This result is expected since the (D,P) values go up, the network become more complex and deeper and performs slightly better on this data set. Also, due to high momentum coefficient; we see a steady decrease for the Cross Entropy Losses for all networks. And accuracy is going up in a stable behaviour and converges to maximum of 34% for the network with (D, P) = (32, 256). To conclude, the network with (D, P) values of (32, 256) performs the best.

Part b

In this part, we are asked to randomly select 5 sampled from the test data and feed the data to trained network. For this, I have used the network with (D, P) = (32, 256). Then, we are asked to list the top 10 candidates for the fourth word. For that, inside MLP class, predict_10 function is created. It is basically randomly selects 5 samples from the test data and lists the top 10 predictions. However, it should be noted that the listed top 10 predictions are not ordered. So, it does not mean the first word in the top 10 list is the word with highest probability of being the 4th word. All top 10 words are randomly ordered since numpy.argpartition is used. The pyhon code for the predict_10 class is given below:

[16]:

```
def predict_10(self, y, y_pred,k,seed):

        '''
        predict_10 function lists the top 10 candidates for the selected triagram.
        INPUTS:

            y            : y is the labels for our data.
            y_pred       : y_pred is the network's prediction.
            k            : K is the number of predictions.
            seed         : For the random initialization.

        RETURNS:

            indeces           : indeces is the indeces of the selected 5 samples.
            indeces_test      : indeces_test is the labels of the selected 5␣
↪sampled from test data.
            indeces_test_pred : indeces_test_pred is the prediction of the network␣
↪for the selected 5 samples
```

```
                                from the test data.
        '''

        np.random.seed(seed)
        indeces = np.random.randint(250, size=5)
        y_new = y[indeces]
        y_pred_new = y_pred[indeces]
        indeces_test_pred = list()
        indeces_test = list()

        for i in range(np.shape(y_pred_new)[0]):
            indeces_test.append(np.argmax(y[i], axis=-1))
            indeces_test_pred.append(np.argpartition(y_pred_new[i], -k, axis=-1)[-k:
 ↪])

        return indeces, indeces_test, indeces_test_pred
```

[16]:
```
# Part B

test_data_1H = np.sum(test_data, axis=-1)

random_indeces, indeces_test, indeces_test_pred = neuralNet2.
 ↪predict_10(test_labels, neuralNet2.forward(test_data_1H), 10, 6)
```

[20]:
```
indeces_test_pred = np.asarray(indeces_test_pred)
three_words = testdata[random_indeces]
the_label = testlbls[random_indeces]

three_words[:] -= 1
the_label[:] -= 1

words_test_pred = list()
for i in range (np.shape(indeces_test_pred)[0]):
    words_test_pred.append(words[indeces_test_pred[i]])


for i in range(np.shape(words_test_pred)[0]):
    ws = [w.decode('UTF-8') for w in words[three_words[i]]]
    print('\033[0m' + "Words are: \033[1m {}".format(ws))
    print('\033[0m' +'The 4th word is : ' + '\033[1m' +str(words[the_label[i]].
 ↪decode('utf-8')))
    print('\033[0m' + 'The predicted top 10 words are: ')
    pred_ws = [words_test_pred[i][j].decode('UTF-8') for j in range(np.
 ↪shape(words_test_pred)[1])]

    print( '\033[1m' + str(pred_ws))
    print('\n')
```

```
Words are:  ['for', 'long', ',']
The 4th word is : he
The predicted top 10 words are:
['you', 'a', 'but', 'they', 'long', 'he', 'she', 'though', 'it', 'i']


Words are:  ['i', 'do', 'not']
The 4th word is : think
The predicted top 10 words are:
['be', 'do', 'like', ',', 'know', 'think', '.', 'want', '?', 'have']


Words are:  ['not', 'that', 'these']
The 4th word is : people
The predicted top 10 words are:
['has', 'would', 'does', '.', 'was', 'did', 'had', 'could', 'years', 'people']


Words are:  ['could', 'you', 'not']
The 4th word is : like
The predicted top 10 words are:
['come', 'do', 'want', ',', 'know', '.', 'think', 'like', 'be', '?']


Words are:  ['know', 'who', 'these']
The 4th word is : people
The predicted top 10 words are:
['says', "'s", 'will', 'did', 'people', 'are', 'is', 'does', 'was', 'can']
```

From the samples, we can see that the actual label and the predictions are really close and for all samples, the label is in the top 10 prediction list. And the predictions and grammar looks logical and reasonable. Thus, it can be concluded that the network is predicting rational and logical words for the 4^th word and is performing well.

Question 3

In this question, we are given 2 demo Python code, one for a fully-connected neural networks and the other for a dropout regularization on such a network. These demo codes will be ran, commented and the inline questions will be answered.

Part a

1 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
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Dropout as a regularizer and Batch/Layer Normalization as a tool to more efficiently optimize deep networks.

[1]:
```
# 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))))
```

[2]:
```
# 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,))
```

2 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:

[3]:
```
# 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:  9.769849468192957e-10
```

3 Affine layer: backward

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

[4]:
```
# 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
```

4 ReLU activation: forward

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

[5]:
```
# 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
```

5 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:

[6]:
```
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
```

5.1 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

5.2 Answer:

Sigmoid and ReLU are the activation functions that has the issue of getting zero or close to zero gradient flow during back-propagation.
For sigmoid, when the sigmoid function value gets too high or too low, the gradient becomes close to zero and the vanishing derivative problem occurs. This problem can occur when the weights and bias initialization is done poorly. Also, for one dimensional case, if the inputs are too small or too large, again the vanishing gradients problem can occur.
For ReLU, if the value of ReLU function is smaller than zero, then the derivative is zero. Thus, a negative input makes the gradient of ReLU zero.

6 "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:

[7]:
```
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:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

7 Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `cs231n/layers.py`.

You can make sure that the implementations are correct by running the following:

[8]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be around the␣
 ↪order of e-9
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should be␣
 ↪around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

8 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.

[9]:
```
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'

# 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: 1.83e-08
W2 relative error: 3.12e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

9 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.

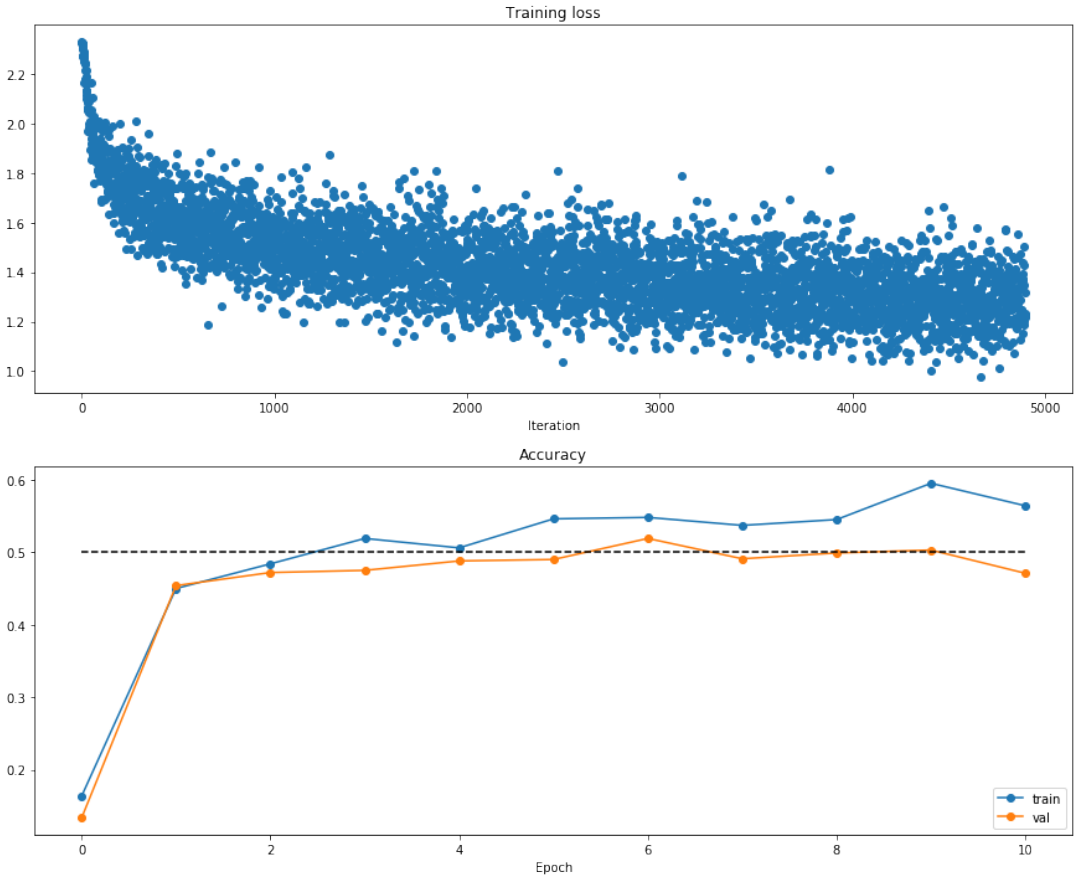
[10]:
```
model = TwoLayerNet()
solver = None

##############################################################################
# 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
(Iteration 301 / 4900) loss: 1.651815
(Iteration 401 / 4900) loss: 1.538214
(Epoch 1 / 10) train acc: 0.450000; val_acc: 0.454000
(Iteration 501 / 4900) loss: 1.608869
(Iteration 601 / 4900) loss: 1.501398
(Iteration 701 / 4900) loss: 1.615213
(Iteration 801 / 4900) loss: 1.656747
(Iteration 901 / 4900) loss: 1.468052
(Epoch 2 / 10) train acc: 0.484000; val_acc: 0.472000
(Iteration 1001 / 4900) loss: 1.505273
(Iteration 1101 / 4900) loss: 1.503323
(Iteration 1201 / 4900) loss: 1.418404
(Iteration 1301 / 4900) loss: 1.356568
(Iteration 1401 / 4900) loss: 1.507079
(Epoch 3 / 10) train acc: 0.519000; val_acc: 0.475000
(Iteration 1501 / 4900) loss: 1.405298
(Iteration 1601 / 4900) loss: 1.425098
(Iteration 1701 / 4900) loss: 1.388389
(Iteration 1801 / 4900) loss: 1.559448
(Iteration 1901 / 4900) loss: 1.469148
(Epoch 4 / 10) train acc: 0.506000; val_acc: 0.488000
(Iteration 2001 / 4900) loss: 1.521458
(Iteration 2101 / 4900) loss: 1.452836
(Iteration 2201 / 4900) loss: 1.515952
(Iteration 2301 / 4900) loss: 1.253438
(Iteration 2401 / 4900) loss: 1.329813
(Epoch 5 / 10) train acc: 0.546000; val_acc: 0.490000
(Iteration 2501 / 4900) loss: 1.385455
(Iteration 2601 / 4900) loss: 1.380330
(Iteration 2701 / 4900) loss: 1.344157
(Iteration 2801 / 4900) loss: 1.516297
(Iteration 2901 / 4900) loss: 1.373451
(Epoch 6 / 10) train acc: 0.548000; val_acc: 0.519000
(Iteration 3001 / 4900) loss: 1.313017
(Iteration 3101 / 4900) loss: 1.139112
(Iteration 3201 / 4900) loss: 1.596601
(Iteration 3301 / 4900) loss: 1.372248
(Iteration 3401 / 4900) loss: 1.524008
(Epoch 7 / 10) train acc: 0.537000; val_acc: 0.491000
(Iteration 3501 / 4900) loss: 1.325397
(Iteration 3601 / 4900) loss: 1.141724
(Iteration 3701 / 4900) loss: 1.368370
(Iteration 3801 / 4900) loss: 1.319290
(Iteration 3901 / 4900) loss: 1.101957
(Epoch 8 / 10) train acc: 0.545000; val_acc: 0.499000
(Iteration 4001 / 4900) loss: 1.239187
(Iteration 4101 / 4900) loss: 1.346376
(Iteration 4201 / 4900) loss: 1.154919
(Iteration 4301 / 4900) loss: 1.073516
(Iteration 4401 / 4900) loss: 1.577285
(Epoch 9 / 10) train acc: 0.595000; val_acc: 0.503000
```

```
(Iteration 4501 / 4900) loss: 1.253220
(Iteration 4601 / 4900) loss: 1.465048
(Iteration 4701 / 4900) loss: 1.484373
(Iteration 4801 / 4900) loss: 1.242994
(Epoch 10 / 10) train acc: 0.564000; val_acc: 0.471000
```

[11]:
```
# 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()
```



10 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.

10.1 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.

[12]:
```
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
```

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.

[13]:
```
# 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()
```

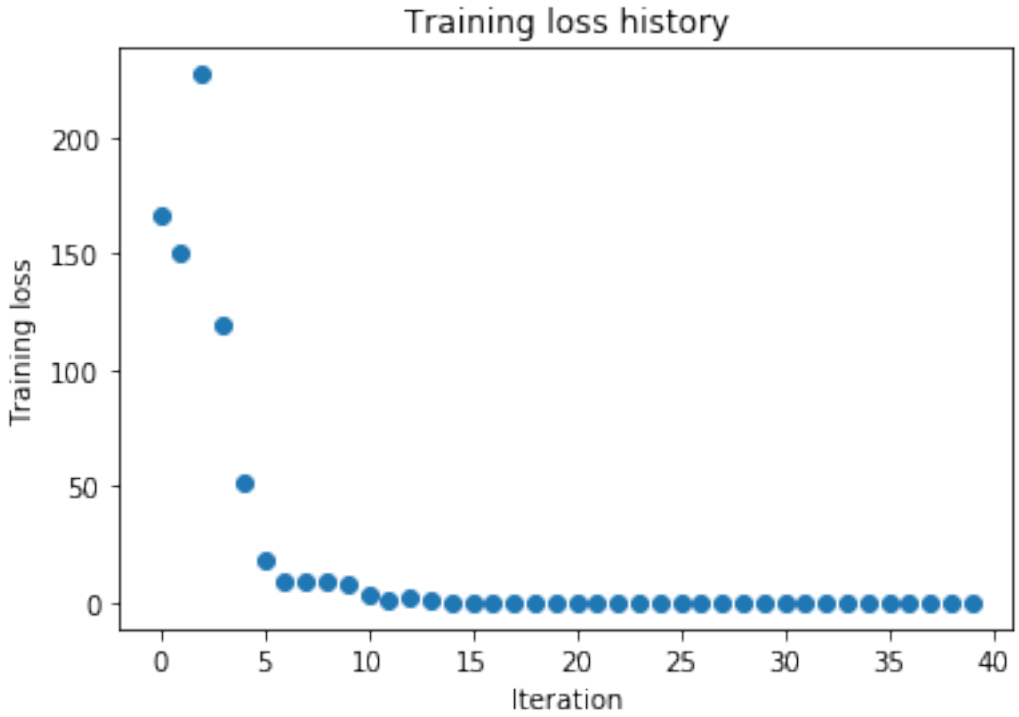
```
(Iteration 1 / 40) loss: 357.428290
(Epoch 0 / 20) train acc: 0.220000; val_acc: 0.111000
(Epoch 1 / 20) train acc: 0.380000; val_acc: 0.141000
(Epoch 2 / 20) train acc: 0.520000; val_acc: 0.138000
(Epoch 3 / 20) train acc: 0.740000; val_acc: 0.130000
(Epoch 4 / 20) train acc: 0.820000; val_acc: 0.153000
(Epoch 5 / 20) train acc: 0.860000; val_acc: 0.175000
(Iteration 11 / 40) loss: 6.726589
(Epoch 6 / 20) train acc: 0.940000; val_acc: 0.163000
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.166000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.164000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.162000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.162000
(Iteration 21 / 40) loss: 0.800243
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.158000
(Iteration 31 / 40) loss: 0.000000
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.158000
```

Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, but you should be able to achieve 100% training accuracy within 20 epochs.

[14]:
```
# 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
(Epoch 6 / 20) train acc: 0.840000; val_acc: 0.122000
(Epoch 7 / 20) train acc: 0.920000; val_acc: 0.113000
(Epoch 8 / 20) train acc: 0.940000; val_acc: 0.125000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.125000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.121000
(Iteration 21 / 40) loss: 0.039138
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.123000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.121000
(Iteration 31 / 40) loss: 0.000644
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.121000
```



10.2 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?

10.3 Answer:

The training with 5 layer network is more difficult than training with 3 layer network. Because with 5 layer net, all of the hyper-parameters and weight initialization should be done much more precisely. With 5 layer network, since there are more matrix multiplication with weights, it is more likely to come across diminishing gradients(weights become too small) or gradient explosion(weights become too large). Thus, the deeper the network , the more precise the hyper-parameter tuning and weight initialization should be.

11 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.

12 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.

[15]:
```
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']))
```

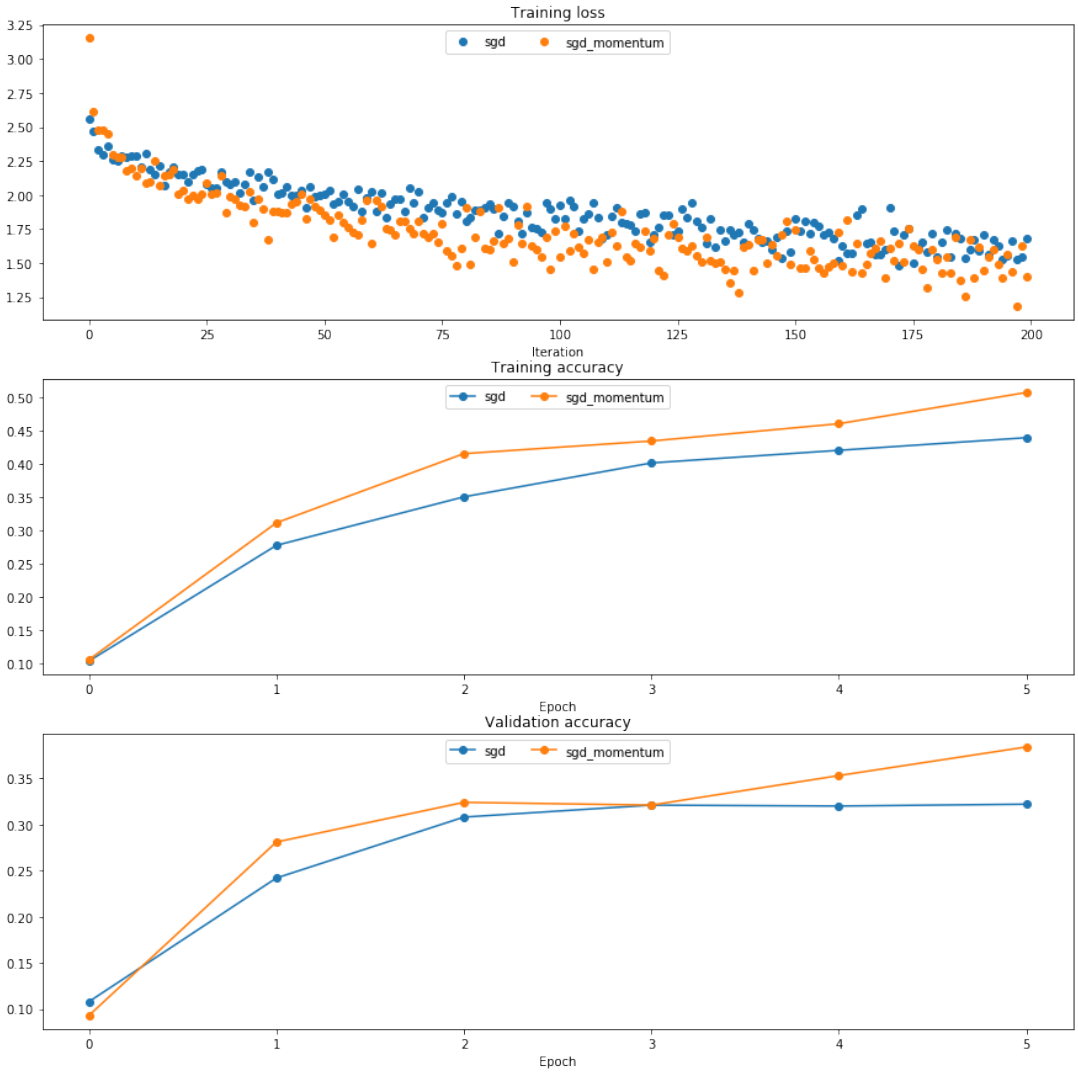
```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

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.

[16]:
```
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)

  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  sgd
(Iteration 1 / 200) loss: 2.559978
```

```
(Epoch 0 / 5) train acc: 0.103000; val_acc: 0.108000
(Iteration 11 / 200) loss: 2.291086
(Iteration 21 / 200) loss: 2.153591
(Iteration 31 / 200) loss: 2.082694
(Epoch 1 / 5) train acc: 0.277000; val_acc: 0.242000
(Iteration 41 / 200) loss: 2.004171
(Iteration 51 / 200) loss: 2.010409
(Iteration 61 / 200) loss: 2.024528
(Iteration 71 / 200) loss: 2.024628
(Epoch 2 / 5) train acc: 0.350000; val_acc: 0.308000
(Iteration 81 / 200) loss: 1.804535
(Iteration 91 / 200) loss: 1.917276
(Iteration 101 / 200) loss: 1.923032
(Iteration 111 / 200) loss: 1.707939
(Epoch 3 / 5) train acc: 0.401000; val_acc: 0.321000
(Iteration 121 / 200) loss: 1.704839
(Iteration 131 / 200) loss: 1.766843
(Iteration 141 / 200) loss: 1.788663
(Iteration 151 / 200) loss: 1.828742
(Epoch 4 / 5) train acc: 0.420000; val_acc: 0.320000
(Iteration 161 / 200) loss: 1.628797
(Iteration 171 / 200) loss: 1.902930
(Iteration 181 / 200) loss: 1.542250
(Iteration 191 / 200) loss: 1.711583
(Epoch 5 / 5) train acc: 0.439000; val_acc: 0.322000

running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153777
(Epoch 0 / 5) train acc: 0.105000; val_acc: 0.093000
(Iteration 11 / 200) loss: 2.145874
(Iteration 21 / 200) loss: 2.032562
(Iteration 31 / 200) loss: 1.985849
(Epoch 1 / 5) train acc: 0.311000; val_acc: 0.281000
(Iteration 41 / 200) loss: 1.882354
(Iteration 51 / 200) loss: 1.855372
(Iteration 61 / 200) loss: 1.649133
(Iteration 71 / 200) loss: 1.806432
(Epoch 2 / 5) train acc: 0.415000; val_acc: 0.324000
(Iteration 81 / 200) loss: 1.907840
(Iteration 91 / 200) loss: 1.510681
(Iteration 101 / 200) loss: 1.546872
(Iteration 111 / 200) loss: 1.512047
(Epoch 3 / 5) train acc: 0.434000; val_acc: 0.321000
(Iteration 121 / 200) loss: 1.677301
(Iteration 131 / 200) loss: 1.504686
(Iteration 141 / 200) loss: 1.633253
(Iteration 151 / 200) loss: 1.745081
(Epoch 4 / 5) train acc: 0.460000; val_acc: 0.353000
(Iteration 161 / 200) loss: 1.485411
(Iteration 171 / 200) loss: 1.610417
(Iteration 181 / 200) loss: 1.528331
(Iteration 191 / 200) loss: 1.449159
(Epoch 5 / 5) train acc: 0.507000; val_acc: 0.384000
```



13 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.

[17]:
```
# 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
```

[18]:
```
# 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
```

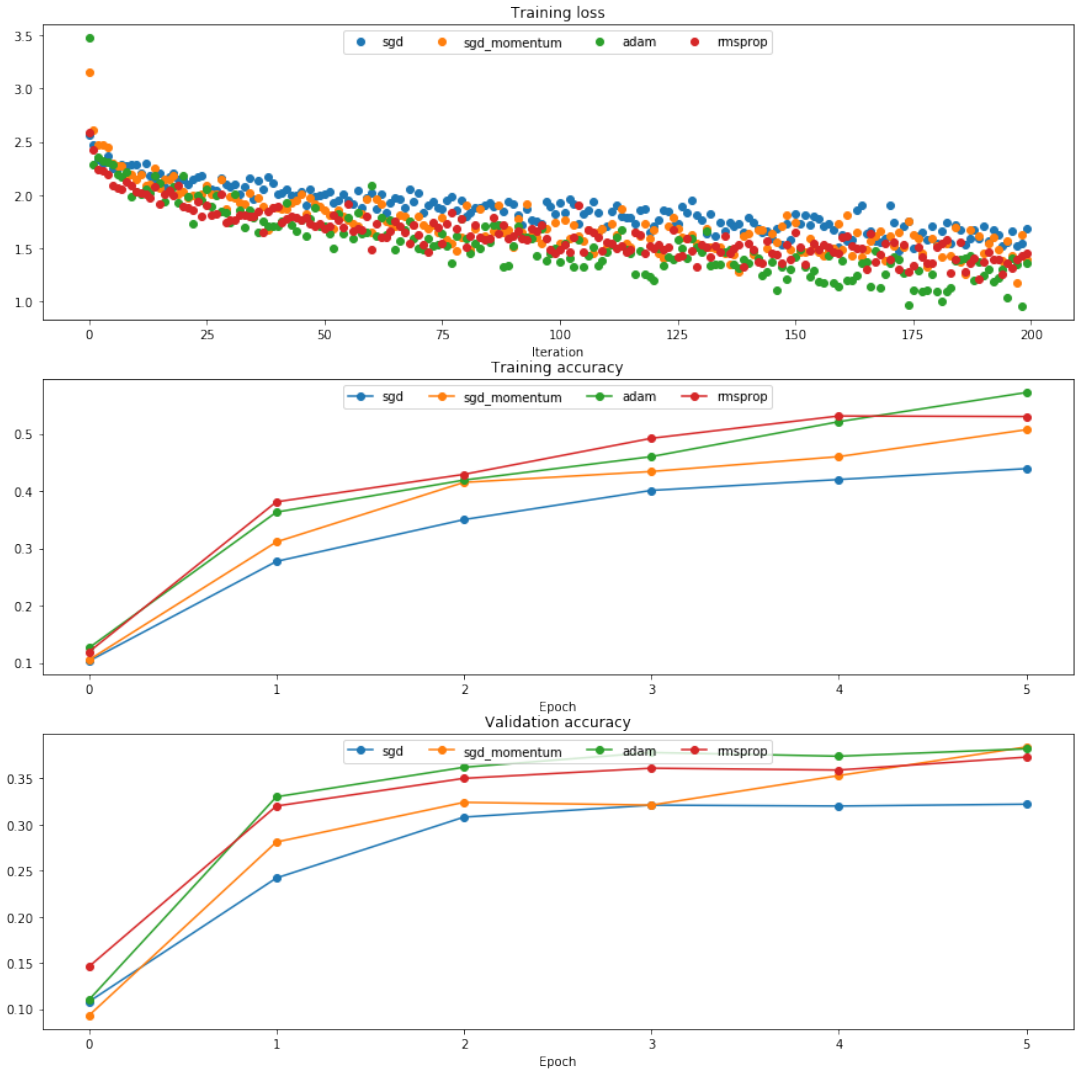
```
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

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

[19]:
```
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
(Iteration 31 / 200) loss: 1.744257
(Epoch 1 / 5) train acc: 0.363000; val_acc: 0.330000
(Iteration 41 / 200) loss: 1.707951
(Iteration 51 / 200) loss: 1.703835
```

```
(Iteration 61 / 200) loss: 2.094758
(Iteration 71 / 200) loss: 1.505558
(Epoch 2 / 5) train acc: 0.419000; val_acc: 0.362000
(Iteration 81 / 200) loss: 1.594429
(Iteration 91 / 200) loss: 1.519017
(Iteration 101 / 200) loss: 1.368522
(Iteration 111 / 200) loss: 1.470400
(Epoch 3 / 5) train acc: 0.460000; val_acc: 0.378000
(Iteration 121 / 200) loss: 1.199064
(Iteration 131 / 200) loss: 1.464705
(Iteration 141 / 200) loss: 1.359863
(Iteration 151 / 200) loss: 1.415069
(Epoch 4 / 5) train acc: 0.521000; val_acc: 0.374000
(Iteration 161 / 200) loss: 1.382818
(Iteration 171 / 200) loss: 1.359900
(Iteration 181 / 200) loss: 1.095947
(Iteration 191 / 200) loss: 1.243088
(Epoch 5 / 5) train acc: 0.572000; val_acc: 0.382000

running with  rmsprop
(Iteration 1 / 200) loss: 2.589166
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.146000
(Iteration 11 / 200) loss: 2.032921
(Iteration 21 / 200) loss: 1.897278
(Iteration 31 / 200) loss: 1.770793
(Epoch 1 / 5) train acc: 0.381000; val_acc: 0.320000
(Iteration 41 / 200) loss: 1.895731
(Iteration 51 / 200) loss: 1.681091
(Iteration 61 / 200) loss: 1.487204
(Iteration 71 / 200) loss: 1.629973
(Epoch 2 / 5) train acc: 0.429000; val_acc: 0.350000
(Iteration 81 / 200) loss: 1.506686
(Iteration 91 / 200) loss: 1.610742
(Iteration 101 / 200) loss: 1.486124
(Iteration 111 / 200) loss: 1.559454
(Epoch 3 / 5) train acc: 0.492000; val_acc: 0.361000
(Iteration 121 / 200) loss: 1.497406
(Iteration 131 / 200) loss: 1.530736
(Iteration 141 / 200) loss: 1.550958
(Iteration 151 / 200) loss: 1.652026
(Epoch 4 / 5) train acc: 0.531000; val_acc: 0.359000
(Iteration 161 / 200) loss: 1.600752
(Iteration 171 / 200) loss: 1.400347
(Iteration 181 / 200) loss: 1.509237
(Iteration 191 / 200) loss: 1.368884
(Epoch 5 / 5) train acc: 0.530000; val_acc: 0.373000
```



13.1 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?

13.2 Answer:

AdaGrad is a per-parameter optimization which assigns small learning rates for parameters related to frequently occurring features and large learning rate for the parameters related to not frequently occurring features. The reason that the updates becoming very small with AdaGrad is that, the squared gradients in the denominator. Since, the term is positive because of the square, it makes the denominator large, leads to a shrinkage of the learning rate. Thus, the updates become very small and the network is learning slowly.

Adam is a a per-parameter optimization like AdaGrad, but Adam is also storing momentum. Thus, updates with the Adam does not become very small like AdaGrad.

14 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.

[20]:
```
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

print("Best val_acc: %f" % best_val_acc)
print("Best weight_scale: %f" % best_weight_scale)
print("Best lr: %f" % best_lr)

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

```
Training...
Weight_scale: 0.010000, lr: 0.000010, val_acc: 0.338000
Weight_scale: 0.010000, lr: 0.000500, val_acc: 0.503000
Weight_scale: 0.010000, lr: 0.000010, val_acc: 0.342000
Weight_scale: 0.020000, lr: 0.000010, val_acc: 0.427000
Weight_scale: 0.020000, lr: 0.000500, val_acc: 0.523000
Weight_scale: 0.020000, lr: 0.000010, val_acc: 0.418000
Weight_scale: 0.005000, lr: 0.000010, val_acc: 0.274000
Weight_scale: 0.005000, lr: 0.000500, val_acc: 0.513000
Weight_scale: 0.005000, lr: 0.000010, val_acc: 0.257000
Best val_acc: 0.523000
Best weight_scale: 0.020000
Best lr: 0.000500
```

15 Test your model!

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

[21]:
```
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:  0.523
Test set accuracy:  0.51
```

Part b

16 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

[1]:
```
# 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
```

[2]:
```
# 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,)
```

17 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.

[3]:
```
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
```

18 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.

[4]:
```
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:  5.44560814873387e-11
```

18.1 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?

18.2 Answer:

In the training, expectation of input x, is the input times the (1-p) since the dropout has a probability (1-p) to reserve the input. However, in the testing, the expectation is itself since it does not get multiplied with the inverse dropout probability. Thus, to keep consistency between the training and test, the values should be divided with the inverse of dropout probability (1-p).

19 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.

[5]:
```
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

Running check with dropout =  0.5
Initial loss:  2.3042759220785896
W1 relative error: 3.11e-07
W2 relative error: 1.84e-08
W3 relative error: 5.35e-08
b1 relative error: 5.37e-09
b2 relative error: 2.99e-09
b3 relative error: 1.13e-10
```

20 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.

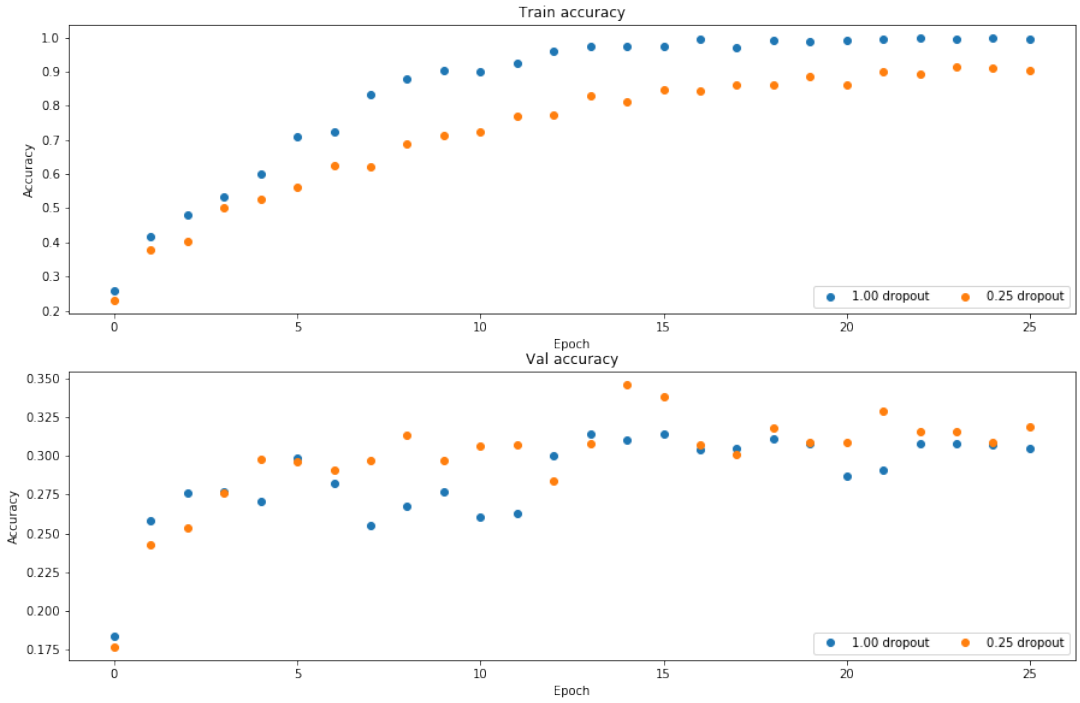
[6]:
```
# Train two identical nets, one with dropout and one without
np.random.seed(231)
num_train = 500
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

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: 7.856643
(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
(Epoch 10 / 25) train acc: 0.898000; val_acc: 0.261000
(Epoch 11 / 25) train acc: 0.924000; val_acc: 0.263000
(Epoch 12 / 25) train acc: 0.960000; val_acc: 0.300000
(Epoch 13 / 25) train acc: 0.972000; val_acc: 0.314000
(Epoch 14 / 25) train acc: 0.972000; val_acc: 0.310000
(Epoch 15 / 25) train acc: 0.974000; val_acc: 0.314000
(Epoch 16 / 25) train acc: 0.994000; val_acc: 0.304000
(Epoch 17 / 25) train acc: 0.970000; val_acc: 0.305000
(Epoch 18 / 25) train acc: 0.990000; val_acc: 0.311000
(Epoch 19 / 25) train acc: 0.988000; val_acc: 0.308000
(Epoch 20 / 25) train acc: 0.992000; val_acc: 0.287000
(Iteration 101 / 125) loss: 0.001417
(Epoch 21 / 25) train acc: 0.994000; val_acc: 0.291000
(Epoch 22 / 25) train acc: 0.998000; val_acc: 0.308000
(Epoch 23 / 25) train acc: 0.996000; val_acc: 0.308000
(Epoch 24 / 25) train acc: 0.998000; val_acc: 0.307000
(Epoch 25 / 25) train acc: 0.994000; val_acc: 0.305000
0.25
(Iteration 1 / 125) loss: 17.318478
(Epoch 0 / 25) train acc: 0.230000; val_acc: 0.177000
(Epoch 1 / 25) train acc: 0.378000; val_acc: 0.243000
(Epoch 2 / 25) train acc: 0.402000; val_acc: 0.254000
(Epoch 3 / 25) train acc: 0.502000; val_acc: 0.276000
(Epoch 4 / 25) train acc: 0.528000; val_acc: 0.298000
(Epoch 5 / 25) train acc: 0.562000; val_acc: 0.296000
(Epoch 6 / 25) train acc: 0.626000; val_acc: 0.291000
(Epoch 7 / 25) train acc: 0.622000; val_acc: 0.297000
(Epoch 8 / 25) train acc: 0.688000; val_acc: 0.313000
(Epoch 9 / 25) train acc: 0.712000; val_acc: 0.297000
(Epoch 10 / 25) train acc: 0.724000; val_acc: 0.306000
(Epoch 11 / 25) train acc: 0.768000; val_acc: 0.307000
```

```
(Epoch 12 / 25) train acc: 0.774000; val_acc: 0.284000
(Epoch 13 / 25) train acc: 0.828000; val_acc: 0.308000
(Epoch 14 / 25) train acc: 0.812000; val_acc: 0.346000
(Epoch 15 / 25) train acc: 0.848000; val_acc: 0.338000
(Epoch 16 / 25) train acc: 0.844000; val_acc: 0.307000
(Epoch 17 / 25) train acc: 0.860000; val_acc: 0.301000
(Epoch 18 / 25) train acc: 0.862000; val_acc: 0.318000
(Epoch 19 / 25) train acc: 0.886000; val_acc: 0.309000
(Epoch 20 / 25) train acc: 0.860000; val_acc: 0.309000
(Iteration 101 / 125) loss: 4.193681
(Epoch 21 / 25) train acc: 0.898000; val_acc: 0.329000
(Epoch 22 / 25) train acc: 0.892000; val_acc: 0.316000
(Epoch 23 / 25) train acc: 0.914000; val_acc: 0.316000
(Epoch 24 / 25) train acc: 0.910000; val_acc: 0.309000
(Epoch 25 / 25) train acc: 0.902000; val_acc: 0.319000
```

[7]:
```
# 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()
```



20.1 Inline Question 2:

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

20.2 Answer:

In the training accuracy, we can see that without dropout the data overfits. With the dropout, the data learns a simpler model and trying to avoid overfit. This can also be confirmed in the validation accuracy, since with the dropout, the accuracy of validation set is slightly better. This shows that the dropout acts as a regularizer and avoid overfitting.

20.3 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)?

20.4 Answer:

The dropout probability is (1-p), and the keep probability is p. As the number of neurons decrease, the p should increase which makes the dropout probability (1-p) small. Because, when the number of neurons diminish, we do not want to drop more neurons since the training of the network will be affected. Thus, when the number of neurons diminish, dropout probability (1-p) should be decreased which means increasing p.