BILKENT UNIVERSITY
ENGINEERING FACULTY
DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

EEE 443/543
Neural Networks – Fall 2021-2022
Assignment 2

Atakan Topcu
21803095

Due 29 November 2021, 17:00 PM

Contents

Question 1 2
Part A 2
Building the class for neural network and layer 2
Optimizing the parameters 6
Part B 8
Part C 8
Part D. 9
Part E 10
Question 2 14
Part A 14
Part B 18
Question 3 20
Part A 20
Part B 20

Question 1

In this question, we are asked to apply binary classification tasks on cat versus car images. The image size is 32x32 and there are two categories. For that reason, we will be using a stochastic mini-batch gradient descent optimization and will be using two separate error metrics given as mean squared error and mean classification error. Error metrics for each epoch will be recorded separately for the training samples and the testing samples. Mean classification error is described as the "percentage of correctly classified images" and this implementation is done according to the given definition.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left(y_{real} - y_{predicted}\right)^2 \; where\; N = \text{Sample Number}, \text{i} = \text{ith sample}$$

Part A

In this part, we will use a backpropagation algorithm and design a multilayered neural network (NN) with a single hidden layer. For this, we will assume a hyperbolic tangent activation function for the network. For this part, we can define two phases. In the first phase, we will construct the neural network through the class functionality of python. In the second phase, we will adjust the parameters such as learning rate and the number of neurons to maximize performance.

Building the class for neural network and layer

For this part, I have first created a class for a single layer. After building a class for a single layer, I have defined a neural network class where I used this class which makes it easy to read the code and implement it. The Layer class can be seen below along with the printed size of the input images.

```
class Layer:
    def __init__(self,inputDim,numNeurons,mean,std,beta):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.beta = beta
        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
        self.weightsAll = np.concatenate((self.weights, self.biases), axis=1)
        self.lastActiv=None
        self.lyrDelta=None
        self.lyrError=None
    def activation(self, x):
        #applying the hyperbolic tangent activation
        x=np.array(x)
        numSamples = x.shape[1]
        tempInp = np.r_[x, [np.ones(numSamples)*-1]]
        self.lastActiv = np.tanh(self.beta*np.matmul(self.weightsAll, tempInp))
        return self.lastActiv

    def activation_derivative(self, x):
        #computing derivative
        return self.beta*(1-(x**2))
```

```
Number of Train Samples: 1900
Train Image Size & Train Label Size: (32, 32, 1900) (1900,)
```

This class enables us to store information of each layer's input dimensions, the number of neurons weights, and biases along with last activation, delta, and error information in a modular way. The weights and biases are initiated with Gaussian random variables with zero means. The standard deviation of the distribution is later submitted from the user as a parameter to be optimized. We were asked to implement hyperbolic tangent activation which is tanh function.

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

I have concatenated the weight and bias vectors in the columns to obtain the matrix referred to as W_{all} (referred as self.weightsAll in the code) and at the same time, we have added a row to the input matrix whose rows are the features and the columns that correspond to the samples. After taking the matrix multiplication of those, I have passed it through tanh activation function.

$$y = \tanh(v)$$
$$v = \boldsymbol{W_{all}} * X_{all}$$
$$where\ \boldsymbol{W_{all}} = [\boldsymbol{W} \mid \text{Bias}], X_{all} = \left[\frac{X}{-1}\right]$$

After the implementation of the activation function, the derivative of the activation is also coded since we will need the derivative of the activation function during backpropagation. The derivative of the activation function is as follows.

$$\frac{\partial \tanh(x)}{\partial x} = \text{sech}^2(x) = 1 - \tanh^2(x)$$

$$\xrightarrow{yields} \frac{\partial \tanh(v)}{\partial x} = \text{sech}^2(v) = 1 - \tanh^2(v) = 1 - y^2$$

This is the reasoning behind saving the last activations of the neurons is that we are already calculating the activated versions in the forward propagation which is computationally efficient. After constructing the layer class, I have moved to define a neural network class that will encompass all the layers to compute forward propagation and backward propagation.

```
class NeuralNetwork:
    def __init__(self):
        self.layers=[]

    def addLayer(self,layer):
        self.layers.append(layer)

    def FowardProp(self,training_inputs):
        #Foward Propagation
        IN=training_inputs
        for layer in self.layers:
            IN=layer.activation(IN)
        return IN
```

Since I am using a list structure, I used an iteration loop to move from one layer to another. After that, I have implemented the forward propagation that uses a for loop to iteratively calculate the activations of each layer. The mathematical expression of the resulting calculation is as follows

$$y_{predicted} = \tanh(\boldsymbol{W_{all,output}} * \dots \tanh\big(\boldsymbol{W_{all,2}} * \tanh\big(\boldsymbol{W_{all,1}} * X_{all}\big)\big))$$

After forward propagation, backward propagation using the delta rule is implemented as individual computing for each layer and each weight becomes computationally intractable as the number of weights increases.

$$MSE = \boldsymbol{E} = \frac{1}{2}\left(Y_{data} - Y_{predicted}\right)^2$$

$$e_{output} = \frac{\partial \boldsymbol{E}}{\partial Y_{predicted}} = Y_{predicted} - Y_{data}$$

$$where\ Y_{predicted}\ is\ the\ output\ of\ the\ network, Y_{data} is\ the\ labels\ of\ each\ sample$$

Then we calculate the delta of the output layer which can be expressed as follows.

$$\delta_{output} = \frac{\partial \tanh(V_{output}(X))}{\partial X} \frac{\partial \boldsymbol{E}}{\partial Y_{predicted}} = \frac{\partial \tanh(V_{output}(X))}{\partial X} e_{output}$$

$$\delta_{output} = \left(1 - Y_{last\ active}{}^2\right)(Y_{predicted} - Y_{data})$$

This concludes the calculations of the output layer, then we move to find the error and delta for the hidden layers.

$e_{hidden} = \boldsymbol{W'^{T}} * \delta'$ *where* prime sign represents the next layer parameters

$$\delta_{hidden} = \frac{\partial \tanh(V_{hidden}(X))}{\partial X} e_{hidden} = \left(1 - Y_{hidden,last\ active}{}^2\right)\boldsymbol{W'^{T}} * \delta'$$

Hence, we now need to consider the update rules for the output and hidden layer structures. This rule is generalized for all the layers with specifying the inputs that will be used for each algorithm.

$$\boldsymbol{W}(n+1) = \boldsymbol{W}(n) + \eta * \delta * \widetilde{X}^{T}$$

$$\widetilde{X} = \begin{cases} X_{last\ activation}, & if\ output \\ \dot{Y}, & if\ hidden \end{cases}$$

$$\dot{Y} = \begin{bmatrix} Y \\ -1 \end{bmatrix}\ where\ Y\ is\ the\ output\ of\ the\ previous\ layer's\ last\ activation$$

Lastly, I have divided the update term by the batch size to avoid saturation. The code for backward propagation is as follows.

```
def BackProp(self,l_rate,batch_size,training_inputs,training_labels):
    #Back Propagation
    foward_out = self.FowardProp(training_inputs)
    for i in reversed(range(len(self.layers))):
        #Output layer
        lyr=self.layers[i]
        if lyr == self.layers[-1]:
            lyr.lyrError=training_labels-foward_out
            derivative=lyr.activation_derivative(lyr.lastActiv)
            lyr.lyrDelta=derivative*lyr.lyrError
        #Other layers
        else:
            nextLyr=self.layers[i+1]
            lyr.lyrError=np.matmul(nextLyr.weightsAll[:,0:nextLyr.weightsAll.shape[1]-1].T, nextLyr.lyrDelta)
            derivative=lyr.activation_derivative(lyr.lastActiv)
            lyr.lyrDelta=derivative*lyr.lyrError

    #UPDATE THE WEIGHT MATRIX
    for i in (range(len(self.layers))):
        lyr=self.layers[i]
        if i==0:
            numSamples = training_inputs.shape[1]
            tempInp = np.r_[training_inputs, [np.ones(numSamples)*-1]]
        else:
            prevLyr=self.layers[i-1]
            numSamples=prevLyr.lastActiv.shape[1],
            tempInp = np.r_[prevLyr.lastActiv, [np.ones(numSamples)*-1]]


        lyr.weightsAll=lyr.weightsAll+l_rate*np.matmul(lyr.lyrDelta, tempInp.T)/batch_size
```

Then I have written the simple function to receive the predictions of the network for error calculation in the training phase. I have modified the labels from 0 and 1 to -1 due to tanh activation output. For the output neuron, I have used one output neuron instead of two since we are dealing with binary classification.

```
def Predict(self,inputIMG):
    out = self.FowardProp(inputIMG)
    out[out>=0] = 1
    out[out<0] = -1
    return out
```

After defining all the necessary functions (forward propagation, backpropagation, etc.), we can now define a final function that will encompass and use all the defined functions. In this final function, it will set up empty sets for MSE, MCE for test and training data. It will store and update the list for each epoch. The code is as follows.

```
def TrainNetwork(self,l_rate,batch_size,training_inputs,training_labels, test_inputs, test_labels, epochNum):
    mseList = []
    mceList = []
    mseTestList = []
    mceTestList = []
    for epoch in range(epochNum):
        print("Epoch:",epoch)
        indexing=np.random.permutation(training_inputs.shape[1])
        #Randomly mixing the samples
        training_inputs=training_inputs[:,indexing]
        training_labels=training_labels[indexing]
        numBatches = int(np.floor(training_inputs.shape[1]/batch_size))
        for j in range(numBatches):
            self.BackProp(l_rate,batch_size,training_inputs[:,j*numBatches:numBatches*(j+1)],\
                          training_labels[j*numBatches:numBatches*(j+1)])

        mse = np.mean((training_labels - self.FowardProp(training_inputs))**2)
        mseList.append(mse)
        mce = np.sum(self.Predict(training_inputs) == training_labels)/len(training_labels)*100
        mceList.append(mce)
        mseT = np.mean((test_labels - self.FowardProp(test_inputs))**2)
        mseTestList.append(mseT)
        mceT = np.sum(self.Predict(test_inputs) == test_labels)/len(test_labels)*100
        mceTestList.append(mceT)
    return mseList, mceList, mseTestList, mceTestList
```

Optimizing the parameters

After building the class for our feed-forward neural network, we will use this class to adjust our parameters for maximizing performance.

```
inputSize = train_images.shape[1]

train_img_flat = train_images.reshape(inputSize**2,train_images.shape[2])
test_img_flat = test_images.reshape(inputSize**2,test_images.shape[2])
print("Train Image after reshaping:",train_img_flat.shape)
print("Test Image after reshaping:",test_img_flat.shape)
train_labels[train_labels == 0] = -1
test_labels[test_labels == 0] = -1
```

```
Train Image after reshaping: (1024, 1900)
Test Image after reshaping: (1024, 1000)
```

```
neuralNet = NeuralNetwork()
neuralNet.addLayer(Layer(inputSize**2, 10, 0, 0.03, 1))
neuralNet.addLayer(Layer(10, 1, 0, 0.03, 1))
```

```
mses, mces, mseTs, mceTs = neuralNet.TrainNetwork(0.25, 57, train_img_flat/255, \
                                                   train_labels, test_img_flat/255, test_labels,400)
```

Images are flattened to use matrix form in our feed-forward calculations and they are normalized as it increases the performance. The chosen parameters can be seen in table 1.

Parameters	Chosen Value
Hidden Neurons	10
Output Neurons	1
Learning Rate	0.25
Batch Size	57
Weights Std.	0.03
Epoch Number	400

Table 1. Chosen Parameter Values.

The error graphs for the test and training MSEs are given below along with the test accuracy.

```
print("Test Accuracy:", str(np.sum(neuralNet.Predict(test_img_flat/255) == test_labels)/len(test_labels)*100) + "%")
```

```
Test Accuracy: 81.6%
```

```
fig, axs = plt.subplots(2, 2)

axs[0, 0].plot(mses)
axs[0, 0].set_title('MSE Over Training')
axs[0, 0].set(ylabel='MSE')

axs[0, 1].plot(mces)
axs[0, 1].set_title('MCE Over Training')
axs[0, 1].set(ylabel='MCE')

axs[1, 0].plot(mseTs)
axs[1, 0].set_title('MSE Over Test')
axs[1, 0].set(xlabel='Epoch', ylabel='MSE')

axs[1, 1].plot(mceTs)
axs[1, 1].set_title('MCE Over Test')
axs[1, 1].set(xlabel='Epoch', ylabel='MCE')
fig.tight_layout(pad=1.0)
```

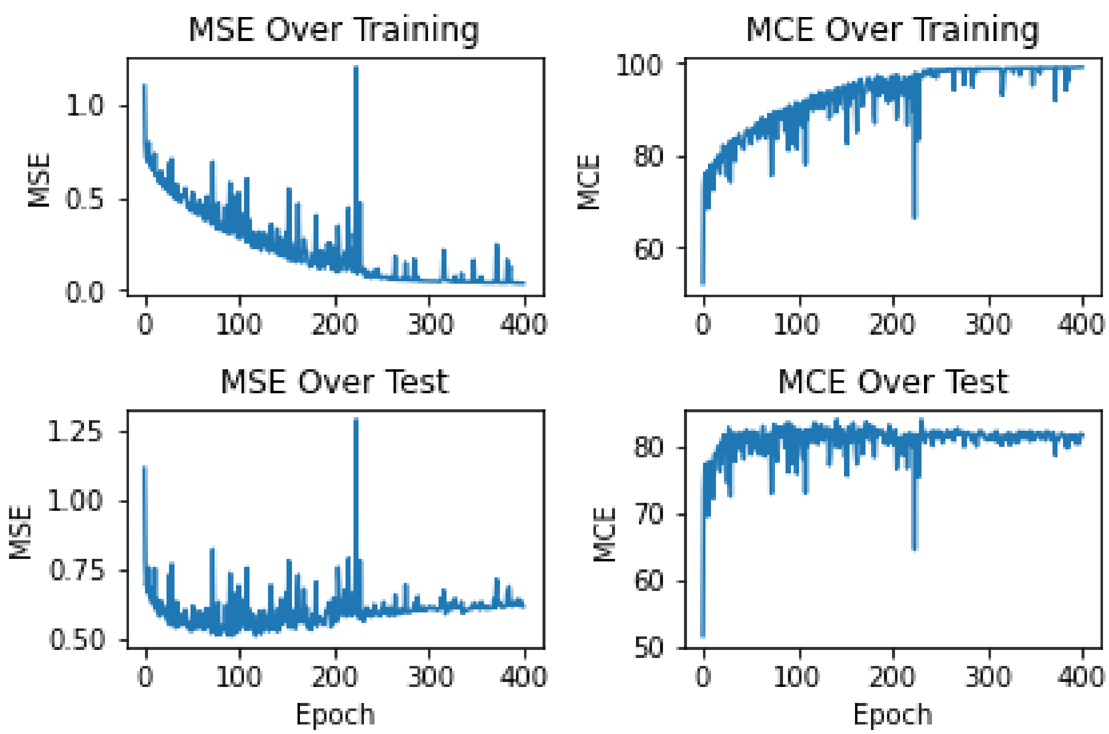


Figure 1. MSE and MCE plots for training and test dataset.

From figure 1, we can see that MSE values converge after some epoch. We see that MSE over training converges to zero while MSE over test converges a value between 0.5 and 075. As for the MCE values, MCE over training converges to 100% while MCE over test converges around 85%. It can be concluded that learning training data perfectly doesn't mean perfect classification for the test data.

Part B

In this part, we are asked to describe the difference in the evolution trend in MSE and MCE for test and training data. We can easily see that MCE is inversely proportional to MSE. As MSE decreases MCE increases. However, there is a difference between MSE and MCE in terms of thresholding. MSE considers the outputs of the network without doing any prediction (thresholding). What this means is that an MSE of a network can be low. However, the predicted value might be far from the actual value. Thus, if the accuracy of the predicted value is needed, MCE is ideal where MSE would not enable us to see this. From figure 1, we can observe that there is a strong correlation between the MCE and MSE since MSE dropping would eventually lead to an increase in the correctly classified classes. However, we can say that while MSEs can be useful in terms of understanding if the algorithm is learning or not, it is not an adequate error metric for a classification task and MCE should be utilized for better understanding the network performance.

Part C

In this part, we are asked to implement, the same neural network algorithm. However, this time, we are asked to use two different hidden neuron numbers to see the effect of neuron numbers on the system. For this reason, I have chosen a significantly bigger neuron number (40) and a significantly lower neuron number (3). After running the algorithm, I have plotted the MSE and MCE for three different cases. The resulting plot can be seen in Figures 2 and 3.

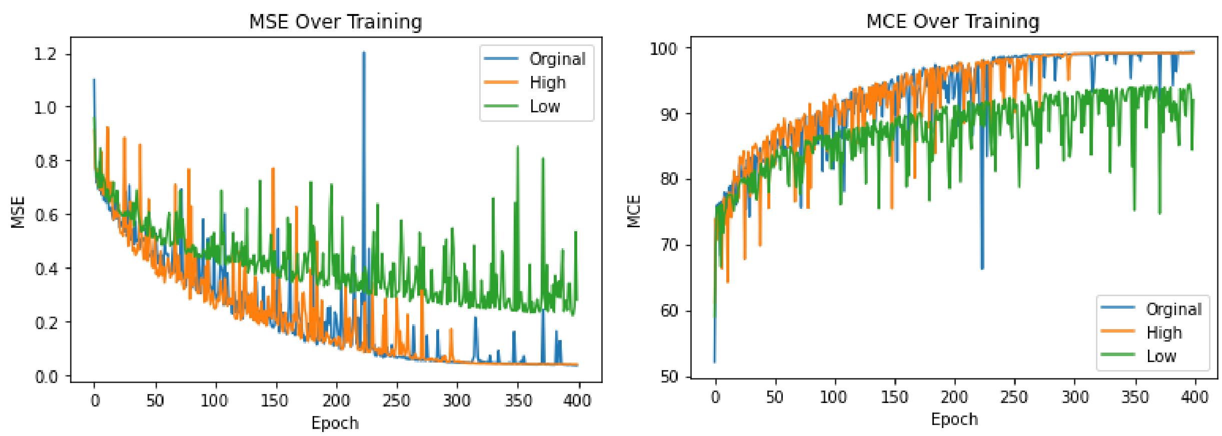


Figure 2. MSE and MCE plots for training dataset for three different hidden neuron number.

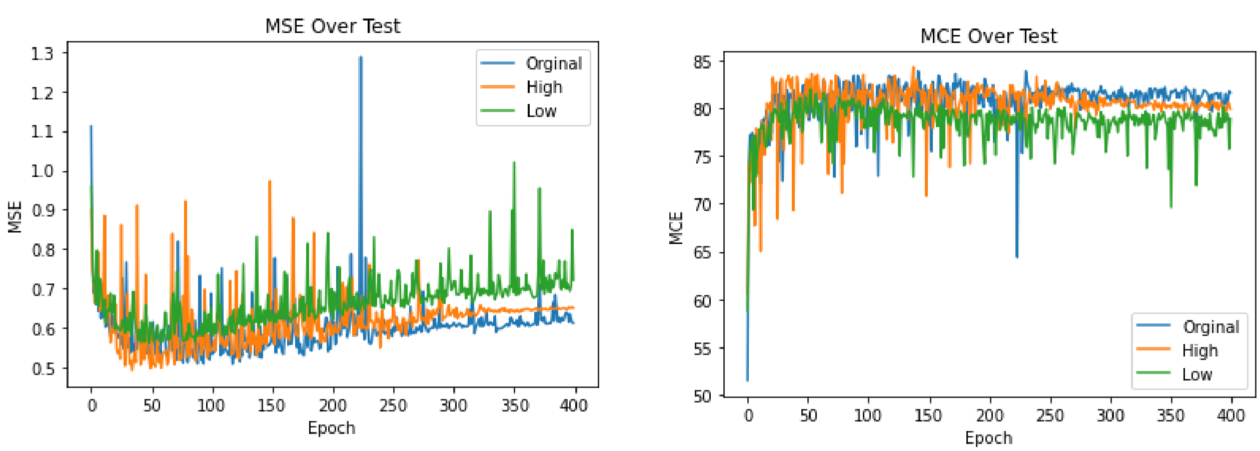


Figure 3. MSE and MCE plots for test dataset for three different hidden neuron number.

We can see that for training data, higher neuron numbers lead to MCE and MSE values similar to our chosen value. However, in test data, due to overfitting and lack of generalization due to increased neuron number, we observe lower MCE and MSE values compared to our chosen neuron number. Low neuron value leads to poor performance on both training and tests MCE, MSE values. So, low neuron number leads to under-fitting and high neuron number leads to over-fitting where both result in poorer performance in both MCE and MSE for the test dataset.

Part D.

In this part, we are asked to design and implement a neural network with two hidden layers. However, the task is the same as before. Since I have already created a class for neural networks and used a modular approach, I have used the same class and added an extra layer.

```
neuralNetTwoHidden = NeuralNetwork()
neuralNetTwoHidden.addLayer(Layer(inputSize**2, 70, 0, 0.03, 1))
neuralNetTwoHidden.addLayer(Layer(70,20, 0, 0.03, 1))
neuralNetTwoHidden.addLayer(Layer(20,1, 0, 0.03, 1))


mses2, mces2, mseTs2, mceTs2 = neuralNetTwoHidden.TrainNetwork(0.3, 57, train_img_flat/255,\
                                                                train_labels, test_img_flat/255, test_labels,220)
```

After initializing the neural network, I have optimized the parameters which can be seen in table 2.

Parameters	Chosen Value
Neurons for the 1st Hidden Layer	70
Neurons for the 2nd Hidden Layer	20
Output Neurons	1
Learning Rate	0.3
Batch Size	57
Weights Std.	0.03
Epoch Number	220

Table 2. Chosen Parameter Values.

The test accuracy of this neural network is around 82.4%. The MSE and MCE plots for the test and training datasets are given below.

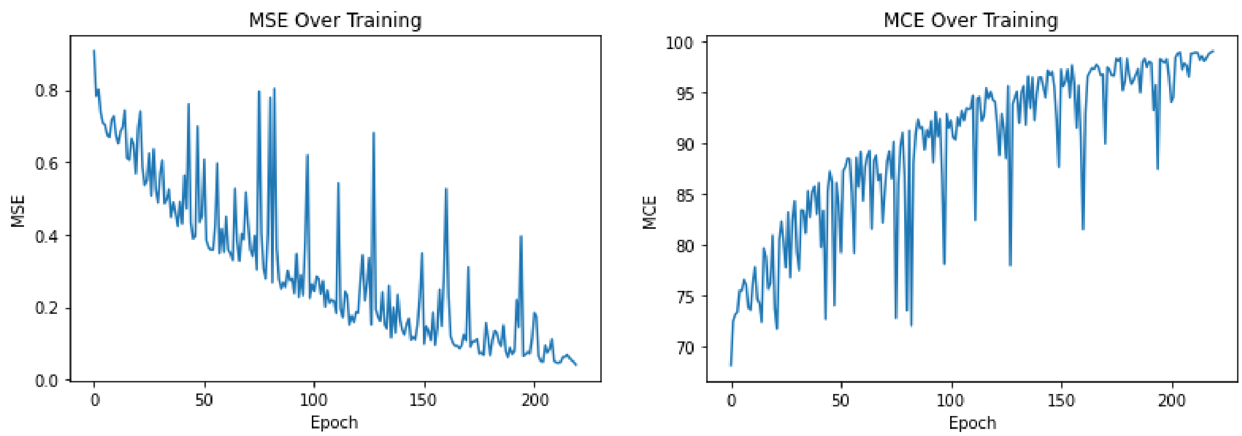


Figure 4. MSE and MCE plots for train dataset for the two hidden layer NN.

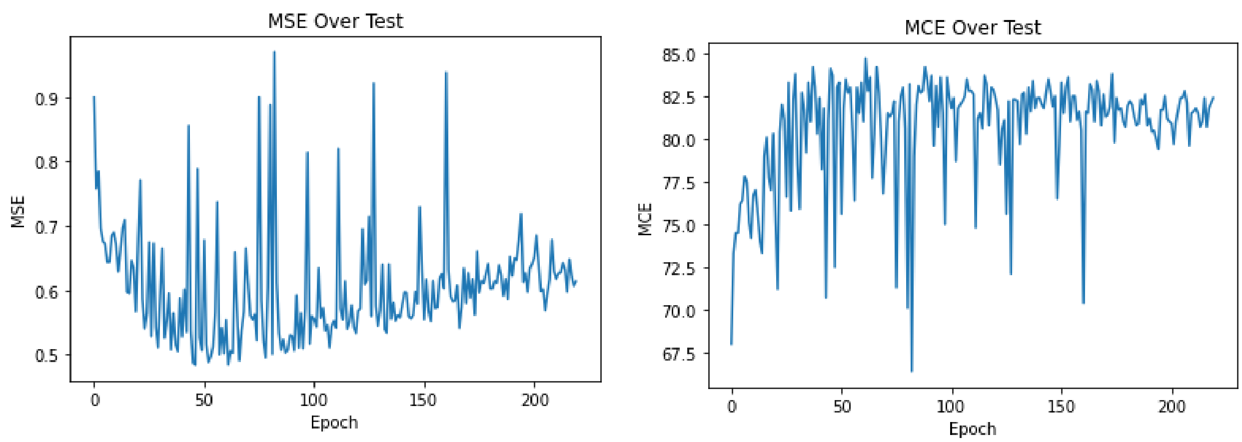


Figure 5. MSE and MCE plots for test dataset for the two hidden layer NN.

By looking at the difference between the network in Part A and the network in Part D, there is an important difference. In the two-layer network, the network has become too complex for the algorithm to learn the pattern in the training as well as the other algorithm. We can say that the complexity of the model has surpassed the complexity of the data. Since the training accuracy couldn't reach 100% as the algorithm in Part A did. Therefore, I can conclude that the one-layer NN has performed better for both MSE and MCE. Furthermore, since the complexity of the network increased compared to the data, the generalization ability of the network has also suffered which can be seen in the MSE and MCE plots of the test data.

Part E

In this part, we are asked to introduce a momentum coefficient to the network and analyze its effects on the MSE and MCE values. In a gradient descent algorithm, the learning rate can affect the time to train the network. Furthermore, as the learning rate increases, the updates will oscillate when approaching a minimum. This creates a very slow learning process. The momentum coefficient enables us to overcome these problems by introducing memory to the system. By doing that learning process becomes faster. Momentum update for weight can be generally written as follows

$$W(n) = -\eta \frac{\mathrm{dE}}{\mathrm{dW}} + \alpha W(n-1), where\ \alpha = momentum\ coeff.$$

$$\xrightarrow{yields} W(n) = -\eta \sum_{k=1}^{n} \alpha^{n-k} \frac{\mathrm{dE(k)}}{\mathrm{dW}}$$

It would be computationally costly to save all the previous gradients in memory. Thus, I applied the following iterative formula. Here, I take into account only the previous update of the layer as a separate term and iteratively update the weight matrix.

$$W(n) = -\eta \frac{\mathrm{dE(n)}}{\mathrm{dW}} + \alpha W(n-1)$$

In order to implement this, I have modified my class. The modified part can be seen below.

```
def BackProp(self,l_rate,batch_size,training_inputs,training_labels,momentCoef):
    #Back Propagation
    foward_out = self.FowardProp(training_inputs)
    for i in reversed(range(len(self.layers))):
        #Output layer
        lyr=self.layers[i]
        if lyr == self.layers[-1]:
            lyr.lyrError=training_labels-foward_out
            derivative=lyr.activation_derivative(lyr.lastActiv)
            lyr.lyrDelta=derivative*lyr.lyrError
        #Other layers
        else:
            nextLyr=self.layers[i+1]
            lyr.lyrError=np.matmul(nextLyr.weightsAll[:,0:nextLyr.weightsAll.shape[1]-1].T, nextLyr.lyrDelta)
            derivative=lyr.activation_derivative(lyr.lastActiv)
            lyr.lyrDelta=derivative*lyr.lyrError

    #UPDATE THE WEIGHT MATRIX
    for i in (range(len(self.layers))):
        lyr=self.layers[i]
        if i==0:
            numSamples = training_inputs.shape[1]
            tempInp = np.r_[training_inputs, [np.ones(numSamples)*-1]]
        else:
            prevLyr=self.layers[i-1]
            numSamples=prevLyr.lastActiv.shape[1],
            tempInp = np.r_[prevLyr.lastActiv, [np.ones(numSamples)*-1]]

        update =  l_rate*np.matmul(lyr.lyrDelta, tempInp.T)/batch_size
        lyr.weightsAll+= update + (momentCoef*lyr.prevUpdate)
        lyr.prevUpdate = update
```

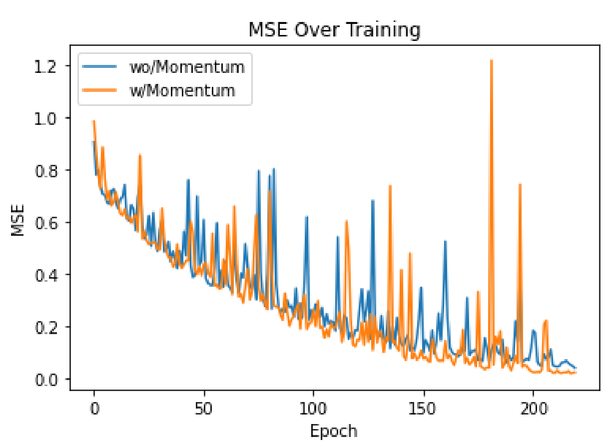
For the backpropagation algorithm, I have changed the weight update part where I introduced the moment coefficient. Furthermore, I have also changed my layer class so that it can now store the previous update of the layer.

```
class LayerWithMomentum:
    def __init__(self,inputDim,numNeurons,mean,std,beta):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.beta = beta
        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
        self.weightsAll = np.concatenate((self.weights, self.biases), axis=1)
        self.lastActiv=None
        self.lyrDelta=None
        self.lyrError=None
        self.prevUpdate = 0
    def activation(self, x):
        #applying the hyperbolic tangent activation
        x=np.array(x)
        numSamples = x.shape[1]
        tempInp = np.r_[x, [np.ones(numSamples)*-1]]
        self.lastActiv = np.tanh(self.beta*np.matmul(self.weightsAll, tempInp))
        return self.lastActiv

    def activation_derivative(self, x):
        #computing derivative
        return self.beta*(1-(x**2))
```

After altering the classes, I have initiated the neural network. I have used the same parameters that I have used for part D to compare the behavior of these two networks.

```
neuralNetTwoHiddenM = NeuralNetworkWithMomentum()
neuralNetTwoHiddenM.addLayer(LayerWithMomentum(inputSize**2, 70, 0, 0.03, 1))
neuralNetTwoHiddenM.addLayer(LayerWithMomentum(70,20, 0, 0.03, 1))
neuralNetTwoHiddenM.addLayer(LayerWithMomentum(20,1, 0, 0.03, 1))
MomentCoef=0.11
mses2M, mces2M, mseTs2M, mceTs2M = neuralNetTwoHiddenM.TrainNetwork(0.3, 57,\
                                    train_img_flat/255, train_labels, test_img_flat/255, test_labels,220,MomentCoef)
```

The accuracy of the network with momentum on the test data has been found as 83.4%. The plots are given below.



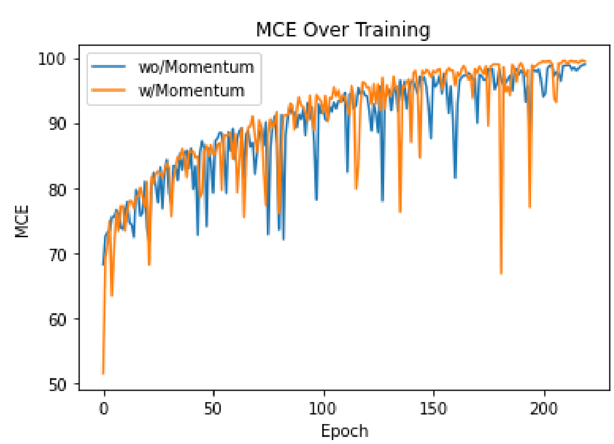


Figure 6. MSE and MCE plots for train dataset for the NN with momentum and without momentum.

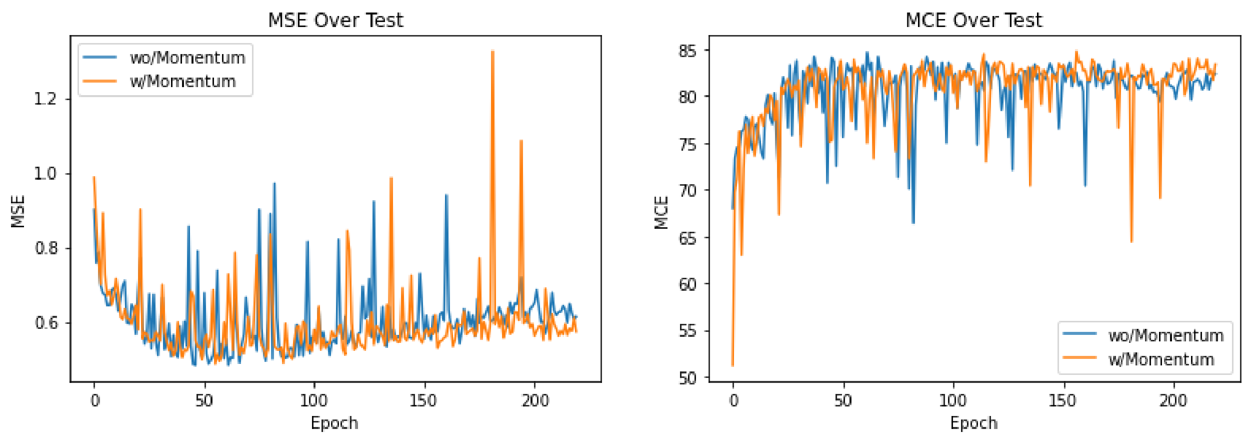


Figure 7. MSE and MCE plots for test dataset for the NN with momentum and without momentum.

From Figures 6 and 7, we can see that MSE and MCE values are more stable as they have lesser high-frequency jumps. Thus, we can comment that momentum acts as a low-pass filter and eliminates high-frequency oscillations. Furthermore, the classification accuracy of the test data turned out to be slightly better. Generally, we observe a more stable convergence.

Question 2

In this question, we are asked to implement a neural network that will take 3 words from the data and output a 4th word that is suitable to these words.

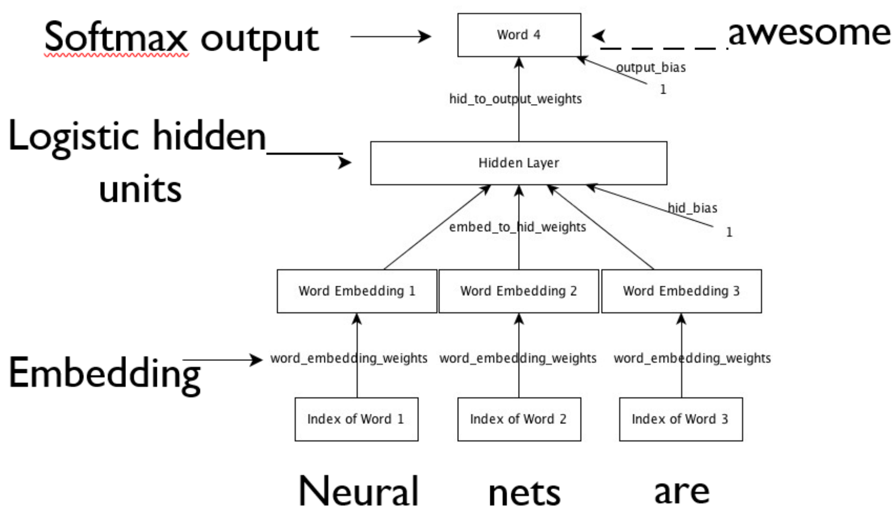


Figure 8. The neural network framework for Question 2.

Different than the first question, this question requires us to implement word embedding to map each word onto a vector of length D. So, the matrix size of the word embedding is Dictionary Size x D. One-hot encoding has been used to convert the indexes given in each word to the vector representation of length Dictionary Size. Furthermore, for each layer, we are asked to use different activation functions. We are required to use sigmoid activations in the other hidden layer and softmax activation for the output layer activation, which are given below.

$$softmax(v_i) = \frac{e^{v_i}}{\sum_{k=1}^{p} e^{v_k}} \ where\ p = \#\ of\ neurons\ of\ the\ layer.$$

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

Part A

This part of the question asks us to implement this network using a set of given parameters that can be optimized for better performance. The given parameters are shown in table 3. For this question, I have used the modified version of the neural network class that I have implemented in the previous question. The modified parts will be explained in the following parts.

Parameters	Chosen Value
Learning Rate	0.15
Batch Size	200
Weights Std.	0.01
Epoch Number	50
Moment Coeff.	0.85
(D,P)	(36,256),(16,128),(8,64)

Table 3. Given Parameter Values.

In order to protect the structural form, I have thought of the Embedding Layer as one form of Layer with a linear activation (f(x)=x) since there is no form of activation mentioned. Furthermore, I have added an activation parameter to the input to be able to use several activation classes with only one class. The code for the constructor is given below.

```
class LayerNLP: #Modified Version of Class in Q1
    def __init__(self,inputDim,numNeurons,mean,std, activation):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation
        if self.activation == 'sigmoid' or self.activation == 'softmax':
            self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
            self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
            self.weightsAll = np.concatenate((self.weights, self.biases), axis=1)
        else:
            self.dictSize = numNeurons
            self.D = inputDim
            self.weights = np.random.normal(mean, std, dictSize*self.D).reshape((dictSize,self.D))

        self.lastActiv=None

        self.lyrDelta=None
        self.lyrError=None
        self.prevUpdate = 0
```

From the _init_ function, we can see that I have considered embedding to be a layer to use the same class for convenience. We can see that we have an if statement that separates the constructed parameters such as the weight matrix. To use the same class as question 1, I have used numNeurons as the dictionary size for the case of embedding matrix. After initialization, I have also altered the activation functions in order to be able to select according to our chosen activation function.

```
    def activationFunction(self, x):
        if(self.activation == 'sigmoid'):
            exp_x = np.exp(2*x)
            return exp_x/(1+exp_x)
        elif(self.activation == 'softmax'):
            exp_x = np.exp(x - np.max(x))
            return exp_x/np.sum(exp_x, axis=0)
        else:
            return x
```

```
    def activationNeuron(self,x):
        if self.activation == 'sigmoid' or self.activation == 'softmax':
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*-1]]
            self.lastActiv = self.activationFunction(np.matmul(self.weightsAll, tempInp))

        else:
            EmbedOut = np.zeros((x.shape[0],x.shape[1], self.D))
            for m in range(EmbedOut.shape[0]): #For each sample
                EmbedOut[m,:,:] = self.activationFunction(np.matmul(x[m,:,:], self.weights))
            EmbedOut = EmbedOut.reshape((EmbedOut.shape[0], EmbedOut.shape[1] * EmbedOut.shape[2]))
            self.lastActiv = EmbedOut.T #For adjusting to other layer's input parameters.
                                        #Otherwise, it will yield error.
        return self.lastActiv

    def activation_derivative(self, x):
        if(self.activation == 'sigmoid'):
            return (x*(1-x))
        elif(self.activation == 'softmax'):
            return x*(1-x)
        else:
            return np.ones(x.shape)
```

Though we have defined the derivative of softmax in the function, it will not be necessary due to the relation of softmax with cross-entropy error which will be explained later.

Now that the layer modification has ended, I have moved to modify the neural network class. Most of the functions in the network remained the same. However, TrainNetwork and BackProp functions are altered.

```
    def BackProp(self,l_rate,batch_size,training_inputs,training_labels,momentCoef):
        foward_out = self.FowardProp(training_inputs)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.lyrDelta=training_labels.T-foward_out
            else:
                nextLyr = self.layers[i+1]
                lyr.lyrError = np.matmul(nextLyr.weights.T, nextLyr.lyrDelta)
                derivative=lyr.activation_derivative(lyr.lastActiv)
                lyr.lyrDelta=derivative*lyr.lyrError


        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                tempInp = training_inputs
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                tempInp = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*-1]]
            if(lyr.activation == 'sigmoid' or lyr.activation == 'softmax'):
                update =  l_rate*np.matmul(lyr.lyrDelta, tempInp.T)/batch_size
                lyr.weightsAll+= update + (momentCoef*lyr.prevUpdate)
            else:
                deltaEmbed = lyr.lyrDelta.reshape((3,batch_size,lyr.D))
                tempInp = np.transpose(tempInp, (1,0,2)) #Rotating the input
                update = np.zeros((tempInp.shape[2], deltaEmbed.shape[2]))
                for i in range(deltaEmbed.shape[0]):
                    update += l_rate * np.matmul(tempInp[i,:,:].T, deltaEmbed[i,:,:])
                update = update/batch_size
                lyr.weights += update + (momentCoef*lyr.prevUpdate)
            lyr.prevUpdate = update
```

From the code, we can see that when we are calculating the deltas, the output layer error is not multiplied by the derivative of the activation function. This is because when we use softmax for the output layer, its derivative does not need to be calculated. The mathematical expression is given below.

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(y_{ic,predict}) = \frac{1}{N}\sum_{i=1}^{N} E \;\; where\; E(i) = -\sum_{c=1}^{C} y_{ic}\log(y_{ic,predict})$$

$$\frac{dE}{dv_j} = -\sum_{c=1}^{C} y_{ic}\frac{\partial \log(y_{ic,predict})}{\partial v_j} = -\sum_{c=1}^{C}\frac{y_{ic}}{y_{ic,predict}}\frac{\partial y_{ic,predict}}{\partial v_j}$$

$$\frac{dE}{dv_j} = -\sum_{c=1}^{C}\frac{y_{ic}}{y_{ic,predict}} y_{ic,predict}\left(1\{c=j\} - y_{ij,predict}\right) = -\sum_{c=1}^{C} y_{ic}\left(1\{c=j\} - y_{ij,predict}\right)$$

$$where\ the\ indicator\ function\ takes\ the\ value\ 1\ when\ c = j, 0\ otherwise.$$

$$\xrightarrow{yields} \sum_{c=1}^{C} y_{ic}\left(y_{ij,predict}\right) - y_{ij} = y_{ij,predict}\sum_{c=1}^{C} y_{ic} - y_{ij} = y_{ij,predict} - y_{ij}$$

```
def TrainNetwork(self,l_rate,batch_size,training_inputs,training_labels, test_inputs, test_labels, epochNum,momentC
    crossList = []
    for epoch in range(epochNum):
        print("Epoch:",epoch)
        indexing=np.random.permutation(len(training_inputs))
        #Randomly mixing the samples
        training_inputs=training_inputs[indexing,:]
        training_labels=training_labels[indexing]
        numBatches = int(np.floor(len(training_inputs)/batch_size))
        for j in range(numBatches):
            train_data_One = SetupData(training_inputs[j*batch_size:batch_size*(j+1),:], dictSize)
            train_labels_One = SetupLabel(training_labels[j*batch_size:batch_size*(j+1)], dictSize)
            self.BackProp(l_rate,batch_size,train_data_One,train_labels_One,momentCoef)

        valOutput = self.FowardProp(test_inputs)
        crossErr = - np.sum(np.log(valOutput) * test_labels.T)/valOutput.shape[1]
        print('Cross-Entropy Error ', crossErr)
        crossList.append(crossErr)

    return crossList
```

After modifying the classes, I have chosen the given parameters. However, for better performance, I have altered some of the parameters. The new parameters can be seen in table 4.

Parameters	Chosen Value
Learning Rate	0.15
Batch Size	200
Weights Std.	0.25
Epoch Number	50
Moment Coeff.	0.7
(D,P)	(36,256),(16,128),(8,64)

Table 4: Chosen Parameter Values.

For every combination of (D, P), I have initiated a new network and trained it. After the training process, the cross-entropy over validation set for each combination has been plotted.

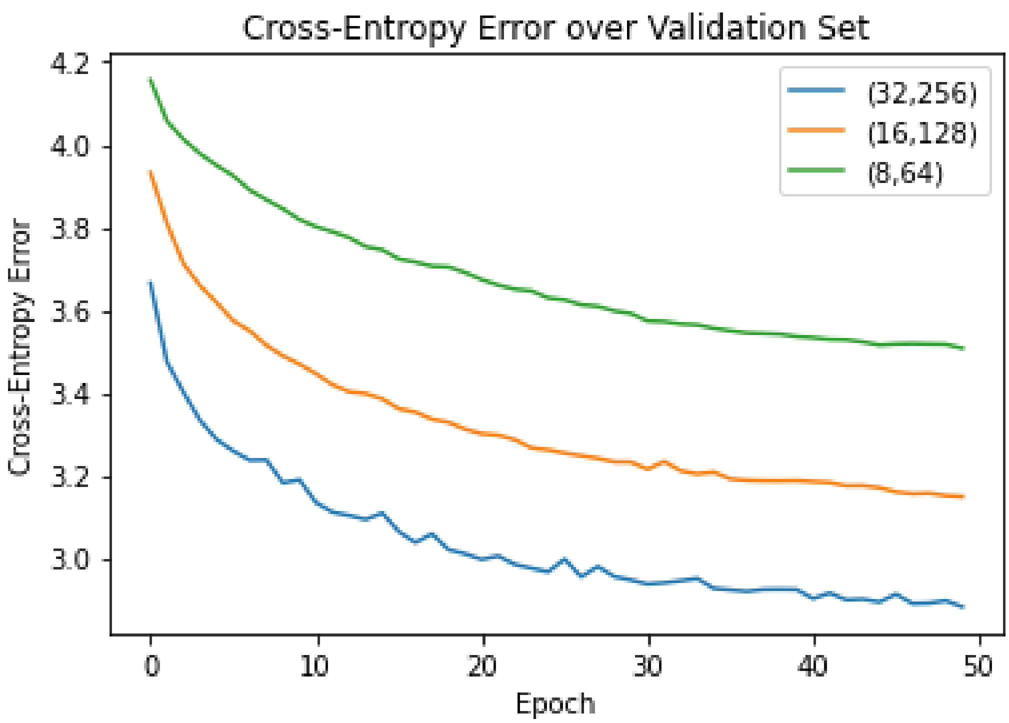


Figure 9. The cross-entropy error of each (D, P).

From figure 9, we can see that after each epoch, the error starts to converge. Furthermore, as we decrease the size of D and P, the cross-entropy error increases. There can be two explanations for this. Firstly, as we decrease the size of D, linear mapping of each word becomes lossier which affects the model's performance. Secondly, since we decrease the hidden neuron number, the network's ability to learn decreases which in turn affects the generalization capability of the network. Because of these two reasons we get the trend that is seen in figure 9.

Part B

In this part, we are asked to pick 5 random samples and generate the 10 highest probability 4^{th}-word output for each of them. For that task, I have created a loop where the first 3 words of the sample are printed along with the top 10 results for the sample and added an extra function for our class.

```
def PredictTopK(self, inputIMG, k):
    out = self.FowardProp(inputIMG)
    return np.argsort(out, axis=0)[:,0:k]
```

This function sorts the 10 highest probability outputs.

```
random_indexes = np.random.permutation(len(test_data))[0:5]

test_samples = test_data[random_indexes,:]
test_outputs = test_labels[random_indexes]

test_samples_One = SetupData(test_samples, 250)

top10 = nn.PredictTopK(test_samples_One, 10)

for i in range(5):
    print('Sample ' + str(i+1)+ ": " + str(words[test_samples[i,0]-1].decode("utf-8"))+' ' + \
          str(words[test_samples[i,1]-1].decode("utf-8"))+' ' \
           + str(words[test_samples[i,2]-1].decode("utf-8")))

    print('The Top 10 predictions: ')
    for j in range(10):
        top = ("["+str(j+1)+". "+ str(words[top10[j,i]-1].decode("utf-8")))+ "]"
        print(top)
```

The output is

```
Sample 1: right there every
The Top 10 predictions:
[1. an]
[2. american]
[3. ,]
[4. members]
[5. big]
[6. two]
[7. right]
[8. more]
[9. 's]
[10. by]
Sample 2: to get this
The Top 10 predictions:
[1. an]
[2. ,]
[3. members]
[4. mr.]
[5. big]
[6. more]
[7. among]
[8. few]
[9. no]
[10. by]
```

```
Sample 3: the money to
The Top 10 predictions:
[1. here]
[2. american]
[3. ,]
[4. if]
[5. former]
[6. would]
[7. see]
[8. by]
[9. members]
[10. because]
Sample 4: for all of
The Top 10 predictions:
[1. ,]
[2. members]
[3. big]
[4. an]
[5. more]
[6. many]
[7. here]
[8. by]
[9. will]
[10. two]
Sample 5: : all right
The Top 10 predictions:
[1. big]
[2. ,]
[3. an]
[4. members]
[5. will]
[6. here]
[7. by]
[8. before]
[9. court]
[10. federal]
```

Here, we can see that among the 10 most probable outputs, some can create a meaningful fourgram such as “right there every american”. However, in some cases, it can create absurd meanings such as “right there every ‘s” Nonetheless, this is a pretty primitive network and thus, despite its flaws, we can say that the network is able to create sensible and logical meanings.

Question 3

In this question, we are asked to run, observe and discuss the notebooks in the given networks named FullyConnectedNets.ipynb and one of either Dropout.ipynb. Some changes applied as scipy.misc does not support imread anymore. Thus, I have used cv2 instead of scipy.misc.

Part A

This part asked us to compile and run a demo for Fully Connected Neural Networks and comment on the results. The networks run on the CIFAR-10 Image Dataset. Overall, this project helped to observe how optimizers modify the performance of the network. For example, with momentum, I have observed how the stochastic gradient descent performance increase converges faster compared to the vanilla stochastic gradient descent. Furthermore, these results are compatible with my results in Question 1 of the assignment. After the moment method, I was also able to see how optimizers such as adam can be implemented in the network and how it alters our learning process. I have concluded that among the chosen optimizers adam provides better learning, generalization capability, and fast convergence. For that reason, I have understood why it is a popular optimizer. The results along with the answers to the inline questions are given on the following pages.

Part B

This part asks us to implement the dropout method to the network that will act as a regularization mechanism that will prevent the network to be over-fitted and increase the generalization of the system. All in all, this project helped me to observe how to use dropout regularization in a neural network. It can be stated that by using the dropout functionality, we have a lesser need of using methods such as L1/L2 regularization where we introduce another term for the penalty. Unlike other regularization methods, dropout offers a very computationally cheap method that can be used for regularization applications. As was the case with the previous part, results along with the answers to the inline questions are given in the following pages.

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:

```python
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:

```python
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.

```python
# 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/1907310/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))))
```

```python
# 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,))
```

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:

```python
# 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:  [illegible]
```

Affine layer: backward

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

```python
# 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:  [illegible]
dw error:  [illegible]
db error:  [illegible]
```

ReLU activation: forward

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

```python
# 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:  [illegible]
```

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:

```python
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:  [illegible]
```

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

Answer:

1. Sigmoid: [illegible]
2. ReLU: [illegible]
3. Leaky ReLU: [illegible]

"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:

```python
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:  [illegible]
dw error:  [illegible]
db error:  [illegible]
```

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:

```python
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:  [illegible]
dx error:  [illegible]

Testing softmax_loss:
loss:  [illegible]
dx error:  [illegible]
```

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.

```python
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,  16.28463190]])
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: [illegible]
W2 relative error: [illegible]
b1 relative error: [illegible]
b2 relative error: [illegible]
Running numeric gradient check with reg =  0.7
W1 relative error: [illegible]
W2 relative error: [illegible]
b1 relative error: [illegible]
b2 relative error: [illegible]
```

Solver

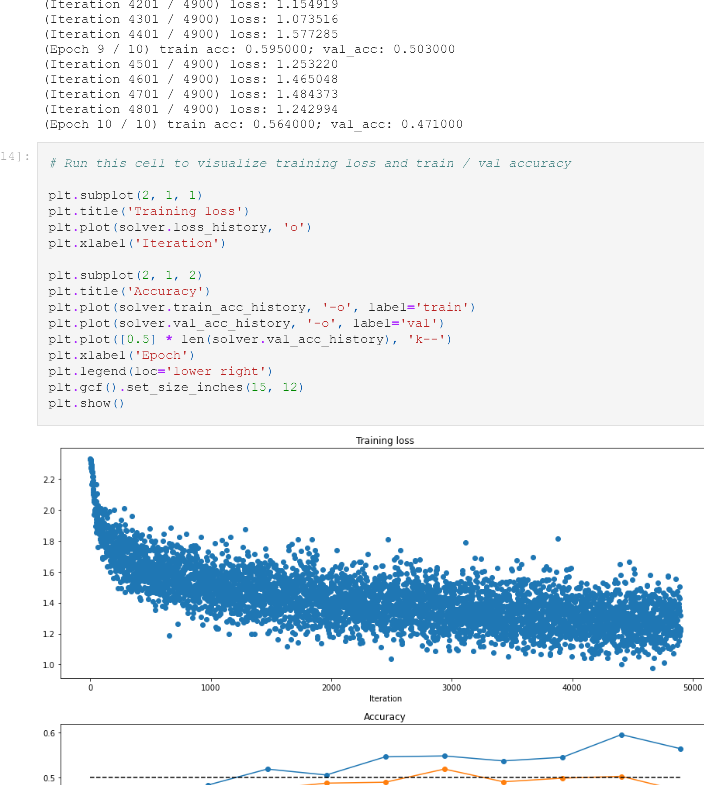
In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

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.

```python
model = TwoLayerNet()
solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                        #
##############################################################################
[illegible]
##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
```

```
[illegible]
```

```python
# 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()
```



Multilayer network

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

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

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

Initial loss and gradient check

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

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

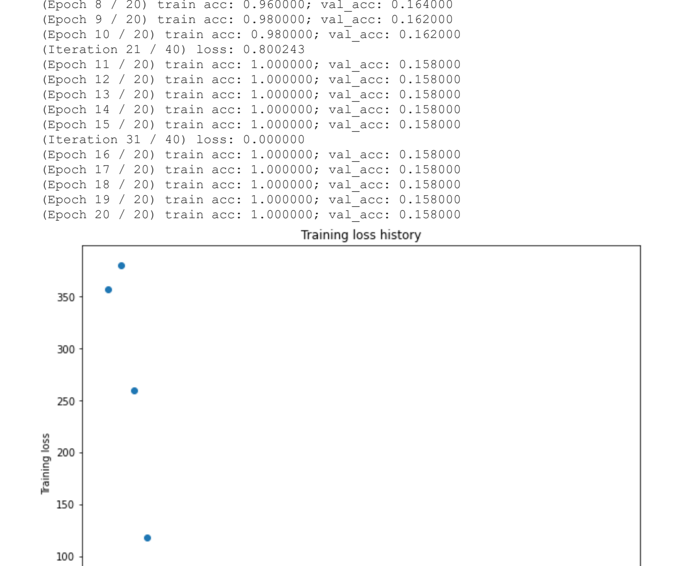
```python
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

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])))
```

```
[illegible]
```

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 weight initialization scale to overfit and achieve 100% training accuracy within 20 epochs.

```python
[illegible]
solver.train()

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

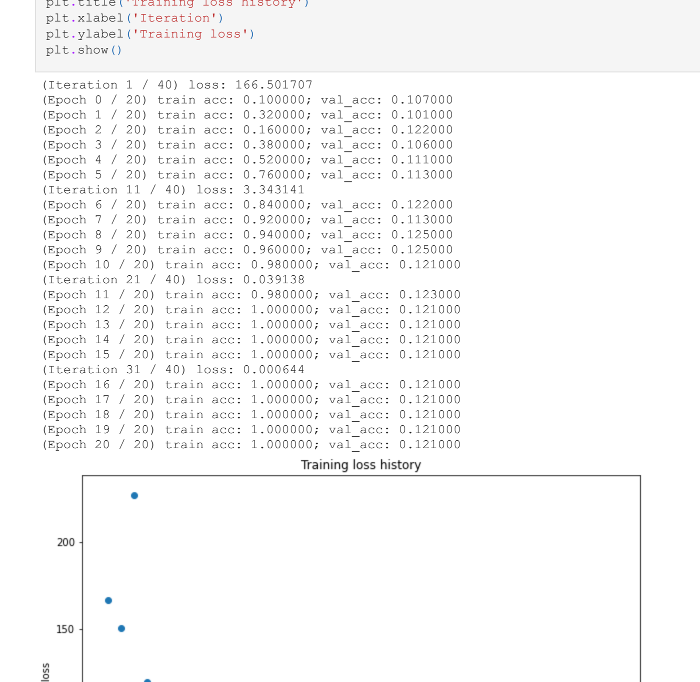
```
[illegible]
```



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 scale, but you should be able to achieve 100% training accuracy within 20 epochs.

```python
[illegible]
)
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
```



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?

Answer:

The three-layer network has better generalization ability to the verification. The five-layer network is more sensitive to parameter initialization, because the five-layer network has more layers and more parameters, and the loss function is more complicated, and it is easier to fall into a local minimum. Furthermore, by increasing the layer number, we reduce the generalization capability of the network which hinders the network's performance on validation sets. As five-layer net initialize weights with higher weight scale, so it needs bigger learning rate. Three-layer net is more robust than five-layer net.

Update rules

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

SGD+Momentum

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

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

In [18]:
```python
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
```

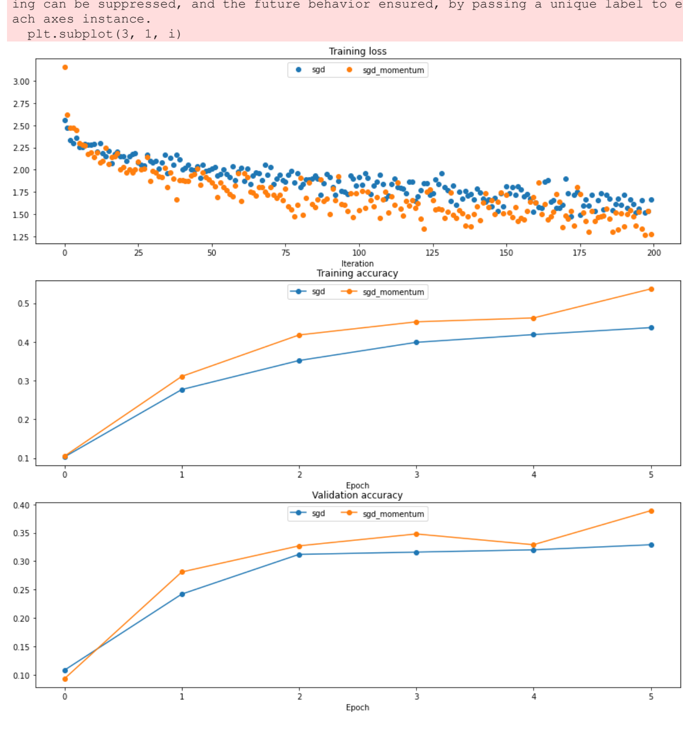
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

In [19]:
```python
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')
```

```python
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: 2.082693
(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: 2.023753
(Iteration 71 / 200) loss: 2.026621
(Epoch 2 / 5) train acc: 0.352000; val_acc: 0.312000
(Iteration 81 / 200) loss: 1.807163
(Iteration 91 / 200) loss: 1.914256
(Iteration 101 / 200) loss: 1.920494
(Iteration 111 / 200) loss: 1.708877
(Epoch 3 / 5) train acc: 0.399000; val_acc: 0.316000
(Iteration 121 / 200) loss: 1.701111
(Iteration 131 / 200) loss: 1.769697
(Iteration 141 / 200) loss: 1.788899
(Iteration 151 / 200) loss: 1.816437
(Epoch 4 / 5) train acc: 0.419000; val_acc: 0.320000
(Iteration 161 / 200) loss: 1.632625
(Iteration 171 / 200) loss: 1.899922
(Iteration 181 / 200) loss: 1.539522
(Iteration 191 / 200) loss: 1.718349
(Epoch 5 / 5) train acc: 0.437000; val_acc: 0.329000

running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153778
(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: 1.985848
(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: 1.806095
(Epoch 2 / 5) train acc: 0.418000; val_acc: 0.327000
(Iteration 81 / 200) loss: 1.910182
(Iteration 91 / 200) loss: 1.498891
(Iteration 101 / 200) loss: 1.541414
(Iteration 111 / 200) loss: 1.518918
(Epoch 3 / 5) train acc: 0.452000; val_acc: 0.348000
(Iteration 121 / 200) loss: 1.625224
(Iteration 131 / 200) loss: 1.525426
(Iteration 141 / 200) loss: 1.589254
(Iteration 151 / 200) loss: 1.717800
(Epoch 4 / 5) train acc: 0.462000; val_acc: 0.329000
(Iteration 161 / 200) loss: 1.629645
(Iteration 171 / 200) loss: 1.483021
(Iteration 181 / 200) loss: 1.425358
(Iteration 191 / 200) loss: 1.363338
(Epoch 5 / 5) train acc: 0.537000; val_acc: 0.389000
```

```
<ipython-input-19-239314d269f9>:39: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 1)
<ipython-input-19-239314d269f9>:42: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 2)
<ipython-input-19-239314d269f9>:45: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 3)
<ipython-input-19-239314d269f9>:49: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, i)
```



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.

```python
# 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
```

In [21]:

```python
# 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
```

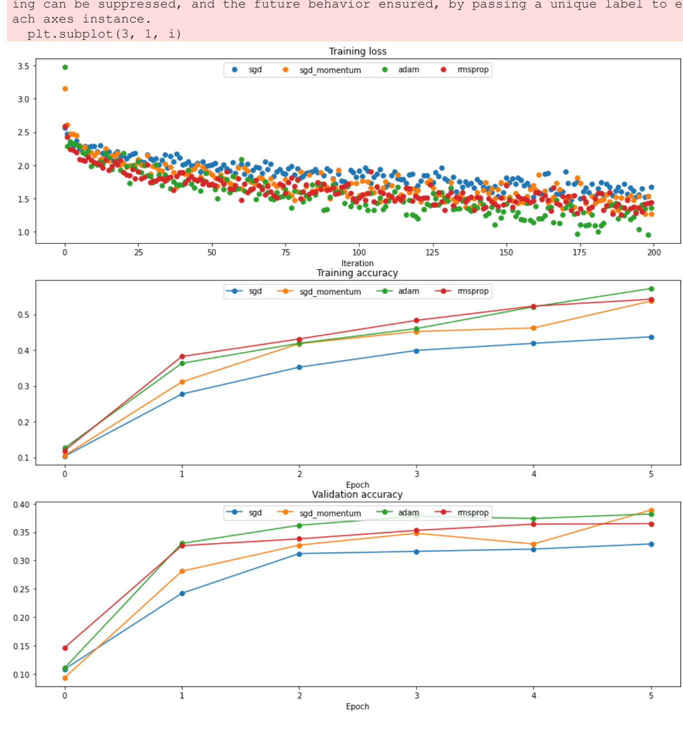
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

In [22]:

```python
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: 1.415068
(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: 2.039570
(Iteration 21 / 200) loss: 1.897350
(Iteration 31 / 200) loss: 1.763338
(Epoch 1 / 5) train acc: 0.382000; val_acc: 0.326000
(Iteration 41 / 200) loss: 1.893851
(Iteration 51 / 200) loss: 1.715728
(Iteration 61 / 200) loss: 1.473747
(Iteration 71 / 200) loss: 1.601207
(Epoch 2 / 5) train acc: 0.431000; val_acc: 0.338000
(Iteration 81 / 200) loss: 1.495041
(Iteration 91 / 200) loss: 1.636391
(Iteration 101 / 200) loss: 1.511503
(Iteration 111 / 200) loss: 1.555876
(Epoch 3 / 5) train acc: 0.483000; val_acc: 0.353000
(Iteration 121 / 200) loss: 1.512182
(Iteration 131 / 200) loss: 1.542606
(Iteration 141 / 200) loss: 1.523618
(Iteration 151 / 200) loss: 1.698618
(Epoch 4 / 5) train acc: 0.523000; val_acc: 0.364000
(Iteration 161 / 200) loss: 1.591356
(Iteration 171 / 200) loss: 1.447764
(Iteration 181 / 200) loss: 1.504678
(Iteration 191 / 200) loss: 1.356730
(Epoch 5 / 5) train acc: 0.542000; val_acc: 0.365000
```

```
<ipython-input-22-c31f2247ce3b>:30: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 1)
<ipython-input-22-c31f2247ce3b>:33: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 2)
<ipython-input-22-c31f2247ce3b>:36: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 3)
<ipython-input-22-c31f2247ce3b>:40: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, i)
```



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?

Answer:

AdaGrad: The cache will become larger and larger after adding the square term of dw, causing the denominator to increase, so that the increment of w decreases every time. For convex functions, it will get slower and slower at the global minimum, which is good. But for non-convex functions, this will be trapped at the local minimum. Because of this, the whole learning process of the network slows down.

For the Adam algorithm, it will not have the same issue as it combines RMSProp with Momentum.

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.

In [23]:

```python
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: 0.341000
Weight_scale: 0.010000, lr: 0.000500, val_acc: 0.499000
Weight_scale: 0.010000, lr: 0.000010, val_acc: 0.342000
Weight_scale: 0.020000, lr: 0.000010, val_acc: 0.425000
Weight_scale: 0.020000, lr: 0.000500, val_acc: 0.521000
Weight_scale: 0.020000, lr: 0.000010, val_acc: 0.421000
Weight_scale: 0.005000, lr: 0.000010, val_acc: 0.274000
Weight_scale: 0.005000, lr: 0.000500, val_acc: 0.512000
Weight_scale: 0.005000, lr: 0.000010, val_acc: 0.257000
Best val_acc: 0.521000
Best weight_scale: 0.020000
Best lr: 0.000500
```

Test your model!

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

In [24]:

```python
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:  0.521
Test set accuracy:  0.506
```

Comments:

Overall, this project helped to observe how optimizers modify the performance of the network. For example, with momentum, I have observed how the stochastic gradient descent performance increase converges faster compared to the vanilla stochastic gradient descent. Furthermore, these results are compatible with my results in Question 1 of the assignment. After moment method, I was also able to see how optimizer such as adam can be implemented in the network and how it alters our learning process. I have concluded that among the chosen optimizers adam provides the better learning, generalization capability and fast convergence. For that reason I have understood why it is a popular optimizer.

In [ ]:

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

In [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
```

In [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,)
```

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.

In [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
```

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.

In [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
```

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?

Answer:

During training, dropout will change the mathematical expectation of input and output. If the input is x, it is retained with probability p, then the expectation is px, because we want to maintain the same mathematical expectation during training and testing, so when forwarding in training, we need to divide by p to ensure input and output. The mathematical expectation remains unchanged.

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.

In [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: 2.58e-08
b2 relative error: 2.99e-09
b3 relative error: 1.13e-10
```

Regularization experiment

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

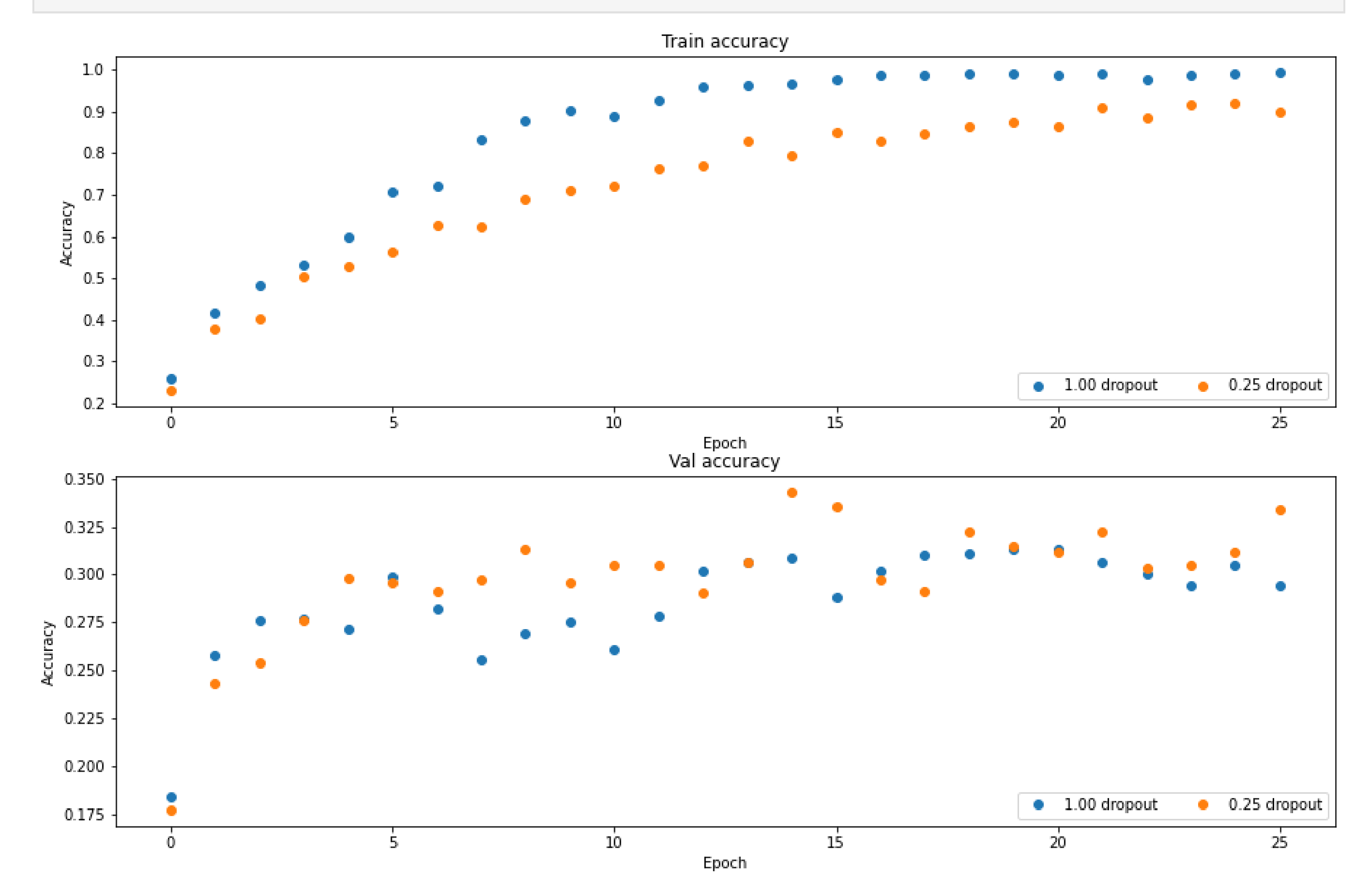
In [6]:

```
# 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: 7.856644
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.276000
(Epoch 3 / 25) train acc: 0.532000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.600000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.708000; val_acc: 0.299000
(Epoch 6 / 25) train acc: 0.722000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.255000
(Epoch 8 / 25) train acc: 0.878000; val_acc: 0.269000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.275000
(Epoch 10 / 25) train acc: 0.888000; val_acc: 0.261000
(Epoch 11 / 25) train acc: 0.926000; val_acc: 0.278000
(Epoch 12 / 25) train acc: 0.960000; val_acc: 0.302000
(Epoch 13 / 25) train acc: 0.964000; val_acc: 0.306000
(Epoch 14 / 25) train acc: 0.966000; val_acc: 0.309000
(Epoch 15 / 25) train acc: 0.976000; val_acc: 0.288000
(Epoch 16 / 25) train acc: 0.988000; val_acc: 0.302000
(Epoch 17 / 25) train acc: 0.988000; val_acc: 0.310000
(Epoch 18 / 25) train acc: 0.990000; val_acc: 0.311000
(Epoch 19 / 25) train acc: 0.990000; val_acc: 0.313000
(Epoch 20 / 25) train acc: 0.988000; val_acc: 0.313000
(Iteration 101 / 125) loss: 0.084169
(Epoch 21 / 25) train acc: 0.990000; val_acc: 0.306000
(Epoch 22 / 25) train acc: 0.978000; val_acc: 0.300000
(Epoch 23 / 25) train acc: 0.986000; val_acc: 0.294000
(Epoch 24 / 25) train acc: 0.990000; val_acc: 0.305000
(Epoch 25 / 25) train acc: 0.994000; val_acc: 0.294000
0.25
(Iteration 1 / 125) loss: 17.318480
(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: 0.690000; val_acc: 0.313000
(Epoch 9 / 25) train acc: 0.712000; val_acc: 0.296000
(Epoch 10 / 25) train acc: 0.722000; val_acc: 0.305000
(Epoch 11 / 25) train acc: 0.764000; val_acc: 0.305000
(Epoch 12 / 25) train acc: 0.770000; val_acc: 0.290000
(Epoch 13 / 25) train acc: 0.830000; val_acc: 0.306000
(Epoch 14 / 25) train acc: 0.794000; val_acc: 0.343000
(Epoch 15 / 25) train acc: 0.850000; val_acc: 0.335000
(Epoch 16 / 25) train acc: 0.830000; val_acc: 0.297000
(Epoch 17 / 25) train acc: 0.848000; val_acc: 0.291000
(Epoch 18 / 25) train acc: 0.866000; val_acc: 0.322000
(Epoch 19 / 25) train acc: 0.874000; val_acc: 0.315000
(Epoch 20 / 25) train acc: 0.866000; val_acc: 0.312000
(Iteration 101 / 125) loss: 4.511463
(Epoch 21 / 25) train acc: 0.910000; val_acc: 0.322000
(Epoch 22 / 25) train acc: 0.886000; val_acc: 0.303000
(Epoch 23 / 25) train acc: 0.916000; val_acc: 0.305000
(Epoch 24 / 25) train acc: 0.920000; val_acc: 0.312000
(Epoch 25 / 25) train acc: 0.900000; val_acc: 0.334000
```

In [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()
```



Inline Question 2:

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

Answer:

When using dropout, the accuracy of training will be lower than compared to the case of not using dropout, and the accuracy of the validation set will be higher than compared to the case of not using dropout, which shows that dropout can be used as regularization. Furthermore, since it can be used as regularization, it reduces the chances of over-fitting the training dataset. Therefore, dropout increases the generalization capability of the network.

Inline Question 3:

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

Answer:

If we need to reduce the size of hidden layers, and also reduce the number of neurons in the hidden layer, then the retention probability of dropout should be increased. If we consider the most extreme case, we reduce the number of neurons in the hidden layer to 1, then if the retention probability of dropout is still very small, then the network is not trained at all most of the time due to lesser proability of using the neuron.

Comments:

All in all this project helped me to observe how to use dropout regularization in a neural network. It can be stated that by using the dropout functionality, we have a lesser need of using methods such as L1/L2 regularization where we introduce another term for penalty. Unlike other regularization methods, dropout offers a very computationally cheap method that can be used for regularization applications.

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import h5py
import seaborn as sn
import sys

question = sys.argv[1]

def Atakan_Topcu_21803095_hw_2(question):
    if question == '1' :
        print("Question", question)
        ##question 1 code goes here
        filename = "assign2_data1.h5"
        file = h5py.File(filename, "r") #Load Data
        # List all groups
        print("Keys: %s" % list(file.keys()))
        testims = list(file.keys())[0]
        testlbls = list(file.keys())[1]
        trainims = list(file.keys())[2]
        trainlbls = list(file.keys())[3]

        # Get the data
        test_images = np.array(file[testims]).T
        test_labels = np.array(file[testlbls]).T
        train_images = np.array(file[trainims]).T
        train_labels = np.array(file[trainlbls]).T

        train_size = train_labels.shape[0]
        print("Number of Train Samples:",train_size) #Shows the number of train
samples

        print("Train Image Size & Train Label Size:",
train_images.shape,train_labels.shape)


        inputSize = train_images.shape[1]

        train_img_flat = train_images.reshape(inputSize**2,train_images.shape[2])
        test_img_flat = test_images.reshape(inputSize**2,test_images.shape[2])
        print("Train Image after reshaping:",train_img_flat.shape)
        print("Test Image after reshaping:",test_img_flat.shape)
        train_labels[train_labels == 0] = -1
        test_labels[test_labels == 0] = -1
```

```
        neuralNet = NeuralNetwork()
        neuralNet.addLayer(Layer(inputSize**2, 10, 0, 0.03, 1))
        neuralNet.addLayer(Layer(10, 1, 0, 0.03, 1))

        mses, mces, mseTs, mceTs = neuralNet.TrainNetwork(0.25, 57,
train_img_flat/255,train_labels, test_img_flat/255, test_labels,400)
        print("Test Accuracy:", str(np.sum(neuralNet.Predict(test_img_flat/255)
== test_labels)/len(test_labels)*100) + "%")


        fig, axs = plt.subplots(2, 2)

        axs[0, 0].plot(mses)
        axs[0, 0].set_title('MSE Over Training')
        axs[0, 0].set(ylabel='MSE')

        axs[0, 1].plot(mces)
        axs[0, 1].set_title('MCE Over Training')
        axs[0, 1].set(ylabel='MCE')

        axs[1, 0].plot(mseTs)
        axs[1, 0].set_title('MSE Over Test')
        axs[1, 0].set(xlabel='Epoch', ylabel='MSE')

        axs[1, 1].plot(mceTs)
        axs[1, 1].set_title('MCE Over Test')
        axs[1, 1].set(xlabel='Epoch', ylabel='MCE')
        fig.tight_layout(pad=1.0)
        plt.show()

        neuralNetHigh = NeuralNetwork()
        neuralNetHigh.addLayer(Layer(inputSize**2, 40, 0, 0.03, 1))
        neuralNetHigh.addLayer(Layer(40,1, 0, 0.03, 1))

        neuralNetLow = NeuralNetwork()
        neuralNetLow.addLayer(Layer(inputSize**2, 3, 0, 0.03, 1))
        neuralNetLow.addLayer(Layer(3, 1, 0, 0.03, 1))

        msesH, mcesH, mseTsH, mceTsH = neuralNetHigh.TrainNetwork(0.25, 57,
train_img_flat/255, train_labels, test_img_flat/255, test_labels,400)
        msesL, mcesL, mseTsL, mceTsL = neuralNetLow.TrainNetwork(0.25, 57,
train_img_flat/255, train_labels, test_img_flat/255, test_labels,400)

        plt.plot(mses)
```

```
        plt.plot(msesH)
        plt.plot(msesL)
        plt.title('MSE Over Training')
        plt.legend(['Orginal', 'High', 'Low'])
        plt.xlabel('Epoch')
        plt.ylabel('MSE')
        plt.show()

        plt.plot(mces)
        plt.plot(mcesH)
        plt.plot(mcesL)
        plt.title('MCE Over Training')
        plt.legend(['Orginal', 'High', 'Low'])
        plt.xlabel('Epoch')
        plt.ylabel('MCE')
        plt.show()

        plt.plot(mseTs)
        plt.plot(mseTsH)
        plt.plot(mseTsL)
        plt.title('MSE Over Test')
        plt.legend(['Orginal', 'High', 'Low'])
        plt.xlabel('Epoch')
        plt.ylabel('MSE')
        plt.show()

        plt.plot(mceTs)
        plt.plot(mceTsH)
        plt.plot(mceTsL)
        plt.title('MCE Over Test')
        plt.legend(['Orginal', 'High', 'Low'])
        plt.xlabel('Epoch')
        plt.ylabel('MCE')
        plt.show()


        neuralNetTwoHidden = NeuralNetwork()
        neuralNetTwoHidden.addLayer(Layer(inputSize**2, 70, 0, 0.03, 1))
        neuralNetTwoHidden.addLayer(Layer(70,20, 0, 0.03, 1))
        neuralNetTwoHidden.addLayer(Layer(20,1, 0, 0.03, 1))


        mses2, mces2, mseTs2, mceTs2 = neuralNetTwoHidden.TrainNetwork(0.3, 57,
train_img_flat/255,train_labels, test_img_flat/255, test_labels,220)
```

```
        print("Test Accuracy:",
str(np.sum(neuralNetTwoHidden.Predict(test_img_flat/255) ==
test_labels)/len(test_labels)*100) + "%")

        plt.plot(mses2)
        plt.title('MSE Over Training')
        plt.xlabel('Epoch')
        plt.ylabel('MSE')
        plt.show()

        plt.plot(mces2)
        plt.title('MCE Over Training')
        plt.xlabel('Epoch')
        plt.ylabel('MCE')
        plt.show()

        plt.plot(mseTs2)
        plt.title('MSE Over Test')
        plt.xlabel('Epoch')
        plt.ylabel('MSE')
        plt.show()

        plt.plot(mceTs2)
        plt.title('MCE Over Test')
        plt.xlabel('Epoch')
        plt.ylabel('MCE')
        plt.show()


        neuralNetTwoHiddenM = NeuralNetworkWithMomentum()
        neuralNetTwoHiddenM.addLayer(LayerWithMomentum(inputSize**2, 70, 0, 0.03,
1))
        neuralNetTwoHiddenM.addLayer(LayerWithMomentum(70,20, 0, 0.03, 1))
        neuralNetTwoHiddenM.addLayer(LayerWithMomentum(20,1, 0, 0.03, 1))
        MomentCoef=0.11
        mses2M, mces2M, mseTs2M, mceTs2M = neuralNetTwoHiddenM.TrainNetwork(0.3,
57,train_img_flat/255, train_labels, test_img_flat/255,
test_labels,220,MomentCoef)
```

```
        print("Test Accuracy:",
str(np.sum(neuralNetTwoHiddenM.Predict(test_img_flat/255) ==
test_labels)/len(test_labels)*100) + "%")


        plt.plot(mses2)
        plt.plot(mses2M)
        plt.legend(['wo/Momentum','w/Momentum'])
        plt.title('MSE Over Training')
        plt.xlabel('Epoch')
        plt.ylabel('MSE')
        plt.show()

        plt.plot(mces2)
        plt.plot(mces2M)
        plt.legend(['wo/Momentum','w/Momentum'])
        plt.title('MCE Over Training')
        plt.xlabel('Epoch')
        plt.ylabel('MCE')
        plt.show()

        plt.plot(mseTs2)
        plt.plot(mseTs2M)
        plt.legend(['wo/Momentum','w/Momentum'])
        plt.title('MSE Over Test')
        plt.xlabel('Epoch')
        plt.ylabel('MSE')
        plt.show()

        plt.plot(mceTs2)
        plt.plot(mceTs2M)
        plt.legend(['wo/Momentum','w/Momentum'])
        plt.title('MCE Over Test')
        plt.xlabel('Epoch')
        plt.ylabel('MCE')
        plt.show()

    elif question == '2' :
        print("Question", question)
        ##question 2 code goes here
        filename = "assign2_data2.h5"
        file = h5py.File(filename, "r") #Load Data

        # List all groups
        print("Keys: %s" % list(file.keys()))
```

```
        testd = list(file.keys())[0]
        testx = list(file.keys())[1]
        traind = list(file.keys())[2]
        trainx = list(file.keys())[3]
        vald = list(file.keys())[4]
        valx = list(file.keys())[5]
        words = list(file.keys())[6]

        # Get the data
        test_labels = np.array(file[testd])
        test_data = np.array(file[testx])
        train_labels = np.array(file[traind])
        train_data = np.array(file[trainx])
        val_labels = np.array(file[vald])
        val_data = np.array(file[valx])
        words = np.array(file[words])

        print("Train Data Size & Train Label Size:",
train_data.shape,train_labels.shape)
        print("Test Data Size & Test Label Size:",
test_data.shape,test_labels.shape)
        print("Validation Data Size & Validation Label Size:",
val_data.shape,val_labels.shape)




        P = 256
        D = 32
        dictSize = 250

        nn = NeuralNetworkNLP()
        nn.addLayer(LayerNLP(D, dictSize,0,0.25, 'embedding'))
        nn.addLayer(LayerNLP(3*D, P, 0,0.25, 'sigmoid'))
        nn.addLayer(LayerNLP(P, dictSize, 0,0.25,'softmax'))

        learnRate = 0.15
        momCoeff = 0.70
        batchSize = 200
        epoch = 50


        val_data_One = SetupData(val_data, dictSize)
        val_labels_One = SetupLabel(val_labels, dictSize)
```

```
        errors = 
nn.TrainNetwork(learnRate,batchSize,train_data,train_labels,val_data_One,val_labe
ls_One,epoch,momCoeff,dictSize)


        P2 = 128
        D2 = 16
        dictSize = 250

        nn2 = NeuralNetworkNLP()
        nn2.addLayer(LayerNLP(D2, dictSize,0,0.25, 'embedding'))
        nn2.addLayer(LayerNLP(3*D2, P2, 0,0.25, 'sigmoid'))
        nn2.addLayer(LayerNLP(P2, dictSize, 0,0.25,'softmax'))

        learnRate = 0.15
        momCoeff = 0.70
        batchSize = 200
        epoch = 50


        val_data_One = SetupData(val_data, dictSize)
        val_labels_One = SetupLabel(val_labels, dictSize)

        errors2 = 
nn2.TrainNetwork(learnRate,batchSize,train_data,train_labels,val_data_One,val_lab
els_One,epoch,momCoeff,dictSize)


        P3 = 64
        D3 = 8
        dictSize = 250

        nn3 = NeuralNetworkNLP()
        nn3.addLayer(LayerNLP(D3, dictSize,0,0.25, 'embedding'))
        nn3.addLayer(LayerNLP(3*D3, P3, 0,0.25, 'sigmoid'))
        nn3.addLayer(LayerNLP(P3, dictSize, 0,0.25,'softmax'))

        learnRate = 0.15
        momCoeff = 0.70
        batchSize = 200
        epoch = 50


        val_data_One = SetupData(val_data, dictSize)
        val_labels_One = SetupLabel(val_labels, dictSize)
```

```
        errors3 =
nn3.TrainNetwork(learnRate,batchSize,train_data,train_labels,val_data_One,val_lab
els_One,epoch,momCoeff,dictSize)


        plt.plot(errors)
        plt.plot(errors2)
        plt.plot(errors3)
        plt.title('Cross-Entropy Error over Validation Set')
        plt.xlabel('Epoch')
        plt.ylabel('Cross-Entropy Error')
        plt.legend(['(32,256)', '(16,128)', '(8,64)'])
        plt.show()


        random_indexes = np.random.permutation(len(test_data))[0:5]

        test_samples = test_data[random_indexes,:]
        test_outputs = test_labels[random_indexes]

        test_samples_One = SetupData(test_samples, 250)

        top10 = nn.PredictTopK(test_samples_One, 10)

        for i in range(5):
            print('Sample ' + str(i+1)+ ": " + str(words[test_samples[i,0]-
1].decode("utf-8"))+' ' +           str(words[test_samples[i,1]-1].decode("utf-
8"))+' '            + str(words[test_samples[i,2]-1].decode("utf-8")))

            print('The Top 10 predictions: ')
            for j in range(10):
                top = ("["+str(j+1)+". "+ str(words[top10[j,i]-1].decode("utf-
8")))+ "]"
                print(top)







class Layer:
    def __init__(self,inputDim,numNeurons,mean,std,beta):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
```

```
        self.beta = beta
        self.weights = np.random.normal(mean,std,
inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std,
numNeurons).reshape(numNeurons,1)
        self.weightsAll = np.concatenate((self.weights, self.biases), axis=1)
        self.lastActiv=None
        self.lyrDelta=None
        self.lyrError=None
    def activation(self, x):
        #applying the hyperbolic tangent activation
        x=np.array(x)
        numSamples = x.shape[1]
        tempInp = np.r_[x, [np.ones(numSamples)*-1]]
        self.lastActiv = np.tanh(self.beta*np.matmul(self.weightsAll, tempInp))
        return self.lastActiv

    def activation_derivative(self, x):
        #computing derivative
        return self.beta*(1-(x**2))




class NeuralNetwork:
    def __init__(self):
        self.layers=[]

    def addLayer(self,layer):
        self.layers.append(layer)

    def FowardProp(self,training_inputs):
        #Foward Propagation
        IN=training_inputs
        for layer in self.layers:
            IN=layer.activation(IN)
        return IN

    def BackProp(self,l_rate,batch_size,training_inputs,training_labels):
        #Back Propagation
        foward_out = self.FowardProp(training_inputs)
        for i in reversed(range(len(self.layers))):
            #Output layer
            lyr=self.layers[i]
            if lyr == self.layers[-1]:
```

```
                lyr.lyrError=training_labels-foward_out
                derivative=lyr.activation_derivative(lyr.lastActiv)

                lyr.lyrDelta=derivative*lyr.lyrError
            #Other layers
            else:
                nextLyr=self.layers[i+1]
                lyr.lyrError=np.matmul(nextLyr.weightsAll[:,0:nextLyr.weightsAll.
shape[1]-1].T, nextLyr.lyrDelta)
                derivative=lyr.activation_derivative(lyr.lastActiv)
                lyr.lyrDelta=derivative*lyr.lyrError

        #UPDATE THE WEIGHT MATRIX
        for i in (range(len(self.layers))):
            lyr=self.layers[i]
            if i==0:
                numSamples = training_inputs.shape[1]
                tempInp = np.r_[training_inputs, [np.ones(numSamples)*-1]]
            else:
                prevLyr=self.layers[i-1]
                numSamples=prevLyr.lastActiv.shape[1],
                tempInp = np.r_[prevLyr.lastActiv, [np.ones(numSamples)*-1]]


            lyr.weightsAll=lyr.weightsAll+l_rate*np.matmul(lyr.lyrDelta,
tempInp.T)/batch_size

    def TrainNetwork(self,l_rate,batch_size,training_inputs,training_labels,
test_inputs, test_labels, epochNum):
        mseList = []
        mceList = []
        mseTestList = []
        mceTestList = []
        for epoch in range(epochNum):
            print("Epoch:",epoch)
            indexing=np.random.permutation(training_inputs.shape[1])
            #Randomly mixing the samples
            training_inputs=training_inputs[:,indexing]
            training_labels=training_labels[indexing]
            numBatches = int(np.floor(training_inputs.shape[1]/batch_size))
            for j in range(numBatches):
                self.BackProp(l_rate,batch_size,training_inputs[:,j*numBatches:nu
mBatches*(j+1)],training_labels[j*numBatches:numBatches*(j+1)])
```

```
            mse = np.mean((training_labels -
self.FowardProp(training_inputs))**2)
            mseList.append(mse)
            mce = np.sum(self.Predict(training_inputs) ==
training_labels)/len(training_labels)*100
            mceList.append(mce)
            mseT = np.mean((test_labels - self.FowardProp(test_inputs))**2)
            mseTestList.append(mseT)
            mceT = np.sum(self.Predict(test_inputs) ==
test_labels)/len(test_labels)*100
            mceTestList.append(mceT)
        return mseList, mceList, mseTestList, mceTestList

    def Predict(self,inputIMG):
        out = self.FowardProp(inputIMG)
        out[out>=0] = 1
        out[out<0] = -1
        return out


class LayerWithMomentum:
    def __init__(self,inputDim,numNeurons,mean,std,beta):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.beta = beta
        self.weights = np.random.normal(mean,std,
inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std,
numNeurons).reshape(numNeurons,1)
        self.weightsAll = np.concatenate((self.weights, self.biases), axis=1)
        self.lastActiv=None
        self.lyrDelta=None
        self.lyrError=None
        self.prevUpdate = 0
    def activation(self, x):
        #applying the hyperbolic tangent activation
        x=np.array(x)
        numSamples = x.shape[1]
        tempInp = np.r_[x, [np.ones(numSamples)*-1]]
        self.lastActiv = np.tanh(self.beta*np.matmul(self.weightsAll, tempInp))
        return self.lastActiv

    def activation_derivative(self, x):
        #computing derivative
        return self.beta*(1-(x**2))
```

```
class NeuralNetworkWithMomentum:
    def __init__(self):
        self.layers=[]

    def addLayer(self,layer):
        self.layers.append(layer)

    def FowardProp(self,training_inputs):
        #Foward Propagation
        IN=training_inputs
        for layer in self.layers:
            IN=layer.activation(IN)
        return IN

    def
BackProp(self,l_rate,batch_size,training_inputs,training_labels,momentCoef):
        #Back Propagation
        foward_out = self.FowardProp(training_inputs)
        for i in reversed(range(len(self.layers))):
            #Output layer
            lyr=self.layers[i]
            if lyr == self.layers[-1]:
                lyr.lyrError=training_labels-foward_out
                derivative=lyr.activation_derivative(lyr.lastActiv)

                lyr.lyrDelta=derivative*lyr.lyrError
            #Other layers
            else:
                nextLyr=self.layers[i+1]
                lyr.lyrError=np.matmul(nextLyr.weightsAll[:,0:nextLyr.weightsAll.
shape[1]-1].T, nextLyr.lyrDelta)
                derivative=lyr.activation_derivative(lyr.lastActiv)
                lyr.lyrDelta=derivative*lyr.lyrError

        #UPDATE THE WEIGHT MATRIX
        for i in (range(len(self.layers))):
            lyr=self.layers[i]
            if i==0:
                numSamples = training_inputs.shape[1]
                tempInp = np.r_[training_inputs, [np.ones(numSamples)*-1]]
            else:
                prevLyr=self.layers[i-1]
```

```
                numSamples=prevLyr.lastActiv.shape[1],
                tempInp = np.r_[prevLyr.lastActiv, [np.ones(numSamples)*-1]]

            update =  l_rate*np.matmul(lyr.lyrDelta, tempInp.T)/batch_size
            lyr.weightsAll+= update + (momentCoef*lyr.prevUpdate)
            lyr.prevUpdate = update

    def TrainNetwork(self,l_rate,batch_size,training_inputs,training_labels,
test_inputs, test_labels, epochNum,momentCoef):
        mseList = []
        mceList = []
        mseTestList = []
        mceTestList = []
        for epoch in range(epochNum):
            print("Epoch:",epoch)
            indexing=np.random.permutation(training_inputs.shape[1])
            #Randomly mixing the samples
            training_inputs=training_inputs[:,indexing]
            training_labels=training_labels[indexing]
            numBatches = int(np.floor(training_inputs.shape[1]/batch_size))
            for j in range(numBatches):
                self.BackProp(l_rate,batch_size,training_inputs[:,j*numBatches:nu
mBatches*(j+1)],training_labels[j*numBatches:numBatches*(j+1)],momentCoef)


            mse = np.mean((training_labels -
self.FowardProp(training_inputs))**2)
            mseList.append(mse)
            mce = np.sum(self.Predict(training_inputs) ==
training_labels)/len(training_labels)*100
            mceList.append(mce)
            mseT = np.mean((test_labels - self.FowardProp(test_inputs))**2)
            mseTestList.append(mseT)
            mceT = np.sum(self.Predict(test_inputs) ==
test_labels)/len(test_labels)*100
            mceTestList.append(mceT)
        return mseList, mceList, mseTestList, mceTestList

    def Predict(self,inputIMG):
        out = self.FowardProp(inputIMG)
        out[out>=0] = 1
        out[out<0] = -1
        return out
```

```
#One-hot encoding
def SetupLabel(y, dictSize):
    out = np.zeros((y.shape[0], dictSize))
    for i in range(y.shape[0]):
        out1 = np.zeros(dictSize)
        out1[y[i]-1] = 1
        out[i,:] = out1
    return out

def SetupData(x, dictSize):
    out = np.zeros((x.shape[0], x.shape[1], dictSize))
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            out1 = np.zeros(dictSize)
            out1[x[i,j]-1] = 1
            out[i,j,:] = out1
    return out




class LayerNLP: #Modified Version of Class in Q1
    def __init__(self,inputDim,numNeurons,mean,std, activation):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation
        if self.activation == 'sigmoid' or self.activation == 'softmax':
            self.weights = np.random.normal(mean,std,
inputDim*numNeurons).reshape(numNeurons, inputDim)
            self.biases = np.random.normal(mean,std,
numNeurons).reshape(numNeurons,1)
            self.weightsAll = np.concatenate((self.weights, self.biases), axis=1)
        else:
            self.dictSize = numNeurons
            self.D = inputDim
            self.weights = np.random.normal(mean, std,
self.dictSize*self.D).reshape((self.dictSize,self.D))

        self.lastActiv=None

        self.lyrDelta=None
        self.lyrError=None
        self.prevUpdate = 0

    def activationFunction(self, x):
```

```
        if(self.activation == 'sigmoid'):
            exp_x = np.exp(2*x)
            return exp_x/(1+exp_x)
        elif(self.activation == 'softmax'):
            exp_x = np.exp(x - np.max(x))
            return exp_x/np.sum(exp_x, axis=0)
        else:
            return x

    def activationNeuron(self,x):
        if self.activation == 'sigmoid' or self.activation ==
'softmax':
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*-1]]
            self.lastActiv = self.activationFunction(np.matmul(self.weightsAll,
tempInp))

        else:
            EmbedOut = np.zeros((x.shape[0],x.shape[1], self.D))
            for m in range(EmbedOut.shape[0]): #For each sample
                EmbedOut[m,:,:] = self.activationFunction(np.matmul(x[m,:,:],
self.weights))
            EmbedOut = EmbedOut.reshape((EmbedOut.shape[0], EmbedOut.shape[1] *
EmbedOut.shape[2]))
            self.lastActiv = EmbedOut.T #For adjusting to other layer's input
parameters.
                                        #Otherwise, it will yield error.
        return self.lastActiv

    def activation_derivative(self, x):
        if(self.activation == 'sigmoid'):
            return (x*(1-x))
        elif(self.activation == 'softmax'):
            return x*(1-x)
        else:
            return np.ones(x.shape)




class NeuralNetworkNLP: #Modified Version of Class in Q1
 def __init__(self):
     self.layers=[]
```

```
 def addLayer(self,layer):
     self.layers.append(layer)

 def FowardProp(self,training_inputs):
     #Foward Propagation
     IN=training_inputs
     for layer in self.layers:
         IN=layer.activationNeuron(IN)
     return IN

 def
BackProp(self,l_rate,batch_size,training_inputs,training_labels,momentCoef):
     foward_out = self.FowardProp(training_inputs)
     for i in reversed(range(len(self.layers))):
         lyr = self.layers[i]
         #outputLayer
         if(lyr == self.layers[-1]):
             lyr.lyrDelta=training_labels.T-foward_out
         else:
             nextLyr = self.layers[i+1]
             lyr.lyrError = np.matmul(nextLyr.weights.T, nextLyr.lyrDelta)
             derivative=lyr.activation_derivative(lyr.lastActiv)
             lyr.lyrDelta=derivative*lyr.lyrError


     #update weights
     for i in range(len(self.layers)):
         lyr = self.layers[i]
         if(i == 0):
             tempInp = training_inputs
         else:
             numSamples = self.layers[i - 1].lastActiv.shape[1]
             tempInp = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*-
1]]
         if(lyr.activation == 'sigmoid' or lyr.activation == 'softmax'):
             update =  l_rate*np.matmul(lyr.lyrDelta, tempInp.T)/batch_size
             lyr.weightsAll+= update + (momentCoef*lyr.prevUpdate)
         else:
             deltaEmbed = lyr.lyrDelta.reshape((3,batch_size,lyr.D))
             tempInp = np.transpose(tempInp, (1,0,2)) #Rotating the input
             update = np.zeros((tempInp.shape[2], deltaEmbed.shape[2]))
             for i in range(deltaEmbed.shape[0]):
                 update += l_rate * np.matmul(tempInp[i,:,:].T,
deltaEmbed[i,:,:])
             update = update/batch_size
```

```
            lyr.weights += update + (momentCoef*lyr.prevUpdate)
        lyr.prevUpdate = update

 def TrainNetwork(self,l_rate,batch_size,training_inputs,training_labels,
test_inputs, test_labels, epochNum,momentCoef,dictSize):
     crossList = []
     for epoch in range(epochNum):
         print("Epoch:",epoch)
         indexing=np.random.permutation(len(training_inputs))
         #Randomly mixing the samples
         training_inputs=training_inputs[indexing,:]
         training_labels=training_labels[indexing]
         numBatches = int(np.floor(len(training_inputs)/batch_size))
         for j in range(numBatches):
             train_data_One =
SetupData(training_inputs[j*batch_size:batch_size*(j+1),:], dictSize)
             train_labels_One =
SetupLabel(training_labels[j*batch_size:batch_size*(j+1)], dictSize)
             self.BackProp(l_rate,batch_size,train_data_One,train_labels_One,mome
ntCoef)

         valOutput = self.FowardProp(test_inputs)
         crossErr = - np.sum(np.log(valOutput) *
test_labels.T)/valOutput.shape[1]
         print('Cross-Entropy Error ', crossErr)
         crossList.append(crossErr)

     return crossList


 def PredictTopK(self, inputIMG, k):
     out = self.FowardProp(inputIMG)
     return np.argsort(out, axis=0)[:,0:k]



Atakan_Topcu_21803095_hw_2(question)
```