BILKENT UNIVERSITY
ENGINEERING FACULTY
DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

EEE 443/543
Neural Networks – Fall 2021-2022
Assignment 3

Atakan Topcu
21803095

Due 30 December 2021, 17:00 PM

Contents

Question 1

In this question, we are asked to implement a single hidden layer autoencoder architecture to extract unsupervised features from images that uses the following loss function.

$$J_{ae} = \frac{1}{2N}\sum_{i=1}^{N}\|d(m) - o(m)\|^2 + \frac{\lambda}{2}\left[\sum_{b=1}^{Lhid}\sum_{a=1}^{Lin}(W_{a,b}^{(1)})^2 + \sum_{c=1}^{Lout}\sum_{b=1}^{Lhid}(W_{b,c}^{(2)})^2\right] + \beta\sum_{b=1}^{Lhid}KL(\rho|\widehat{\rho_b})$$

In this loss function, the first term represents the mean squared error (MSE). The second term L2 (Tikhonov) regularization and the third term represents KL Divergence applied to the mean activations of the hidden layer.

Part A

This part asks us to import the given file and preprocess the images inside the file by converting RGB images to grayscale. The formula for this preprocessing is given below.

$$Y = 0.2126 * \mathrm{R} + 0.7152 * \mathrm{G} + 0.0722 * \mathrm{B}$$

After this conversion process, we are asked to normalize the data by subtracting each image with its pixel intensity mean and clipping the data range between 3 standard deviations. Furthermore, to prevent saturation, we map the clipped data to [0.1 0.9] range. The mapping formula is as follows.

$$s = 0.9 + ((s - \min(s)) * \frac{\max(s) - \min(s)}{2 * \max(s)})$$

The code for preprocessing is as follows.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import h5py
import seaborn as sn
filename = "assign3_data1.h5"
```

```
file = h5py.File(filename, "r") #Load Data
```

```
# List all groups
print("Keys: %s" % list(file.keys()))
data = list(file.keys())[0]
invXForm = list(file.keys())[1]
xForm = list(file.keys())[2]

# Get the data
data = np.array(file[data])
invXForm = np.array(file[invXForm])
xForm = np.array(file[xForm])
```

```
Keys: ['data', 'invXForm', 'xForm']
```

```
print('Data:', data.shape)
```

```
Data: (10240, 3, 16, 16)
```

Part A

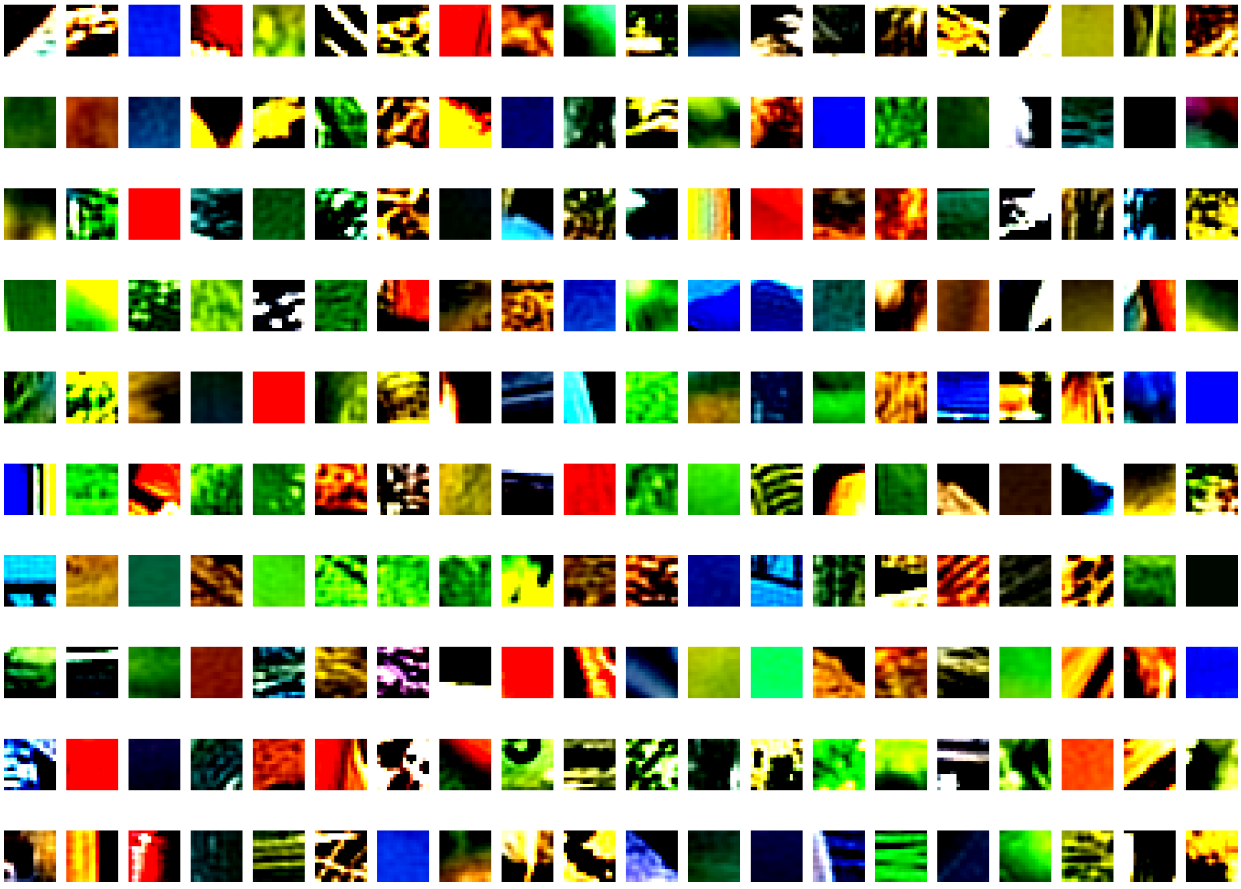
```
PreprocessedData=0.2126*data[:,0,:,:]+0.7152*data[:,1,:,:]+0.0722*data[:,0,:,:]
```

```
print('Preprocessed Data:', PreprocessedData.shape)
```

```
Preprocessed Data: (10240, 16, 16)
```

```
meanIntensity = np.mean(PreprocessedData, axis=(1,2))
NormalizedData = np.zeros(PreprocessedData.shape)
#Mean pixel intensity subtraction
for i in range(PreprocessedData.shape[0]):
        NormalizedData[i,:,:]=PreprocessedData[i,:,:]-meanIntensity[i]
#Standart deviation
std = np.std(NormalizedData)
NormalizedData = np.minimum(NormalizedData, 3*std)
NormalizedData = np.maximum(NormalizedData, -3*std)
#[0.1 0.9] scaling
minRange=0.1
maxRange=0.9
difference=maxRange-minRange
#Adjusting the scale to the range [0.1 0.9]
NormalizedData=minRange+((NormalizedData-np.min(NormalizedData))*difference)/(2*np.max(NormalizedData))
print(np.min(NormalizedData))
print(np.max(NormalizedData))
```

```
0.1
0.9
```



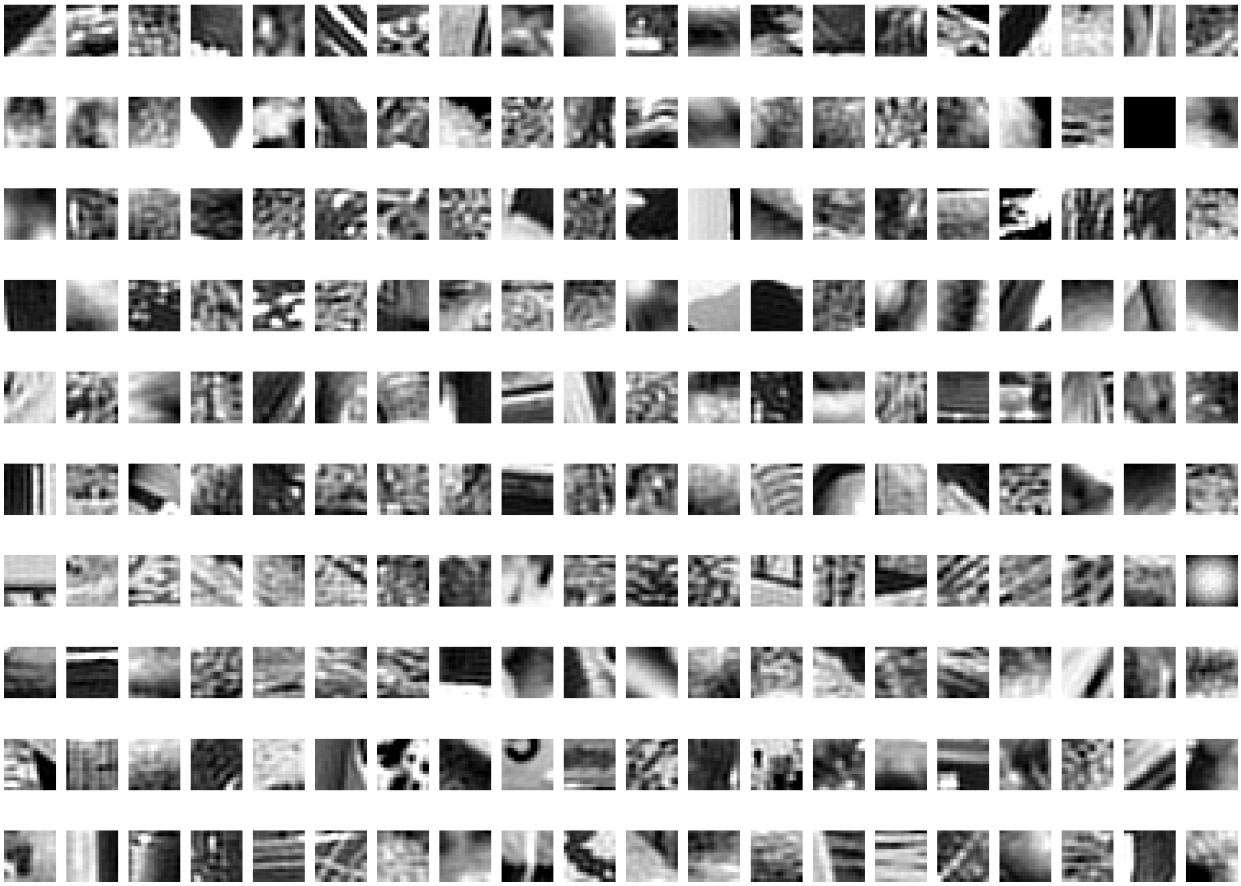


Figure 1, 2. Original 200 sample images with their preprocessed versions.

From figure 1 and 2, we can state that images are correctly preprocessed. However, there can be slight deviations in the resulting images as we have clipped some parts of the images.

Part B

In this part, we are asked to initialize weights before training and then, implement the whole system along with the aeCost function. The initialization is done as follows.

$$r = \sqrt{\frac{6}{Lprev + Lpost}} \xrightarrow{yields} W = [-r\ r], b = [-r\ r]$$

$$W_1 = W_2^T$$

The question gives us already decided variable values for lamda and hidden neuron numbers. Thus, in order to create the network, I have first flattened the preprocessed image and initialized the variables. Furthermore, SGD with full batch has been utilized as they give more robust and accurate results.

Part B

```
#Flatten images to represent each pixel as neuron.
Pixel = NormalizedData.shape[1]
SampleSize=NormalizedData.shape[0]
Flat_Data = np.reshape(NormalizedData, (SampleSize,Pixel**2))
print("Flattened Data Shape:", Flat_Data.shape)
Linput=Loutput=Pixel**2
Lhid = 64
lmb = 5e-4
beta = 0.06
rho = 0.2
params = (Linput, Lhid, lmb, beta, rho)
```

```
Flattened Data Shape: (10240, 256)
```

```
class Autoencoder():
    def w0(self,Lpre,Lpost): #Lpre;post are the number of neurons on either side of the connection weights.
        return np.sqrt(6/(Lpre+Lpost))

    def WeightInitialization(self, Lin, Lhidden, Lout):
        np.random.seed(99)
        W1=np.random.uniform(-self.w0(Lin, Lhidden),self.w0(Lin, Lhidden),(Lin,Lhidden))
        W2=np.random.uniform(-self.w0(Lhidden, Lout),self.w0(Lhidden, Lout),(Lhidden, Lout))
        b1=np.random.uniform(-self.w0(Lin, Lhidden),self.w0(Lin, Lhidden),(1, Lhidden))
        b2=np.random.uniform(-self.w0(Lhidden, Lout),self.w0(Lhidden, Lout),(1, Lout))
        We=(W1, W2, b1, b2)
        return We

    def sigmoid(self,x):
        #Sigmoid Function
        expx = np.exp(x)
        return expx/(1+expx)

    def sigmoidDerivative(self,x):
        return x*(1-x)

    def fowardpass(self,We,data): #Simple Foward Pass
        W1, W2, b1, b2 = We
        W1 = np.concatenate((W1,b1), axis=0)
        W2 = np.concatenate((W2,b2), axis=0)
        data_new = np.concatenate((data, np.ones((data.shape[0], 1))), axis=1)
        hid = self.sigmoid(np.matmul(data_new,W1))
        hid_new = np.concatenate((hid, np.ones((hid.shape[0], 1))), axis=1)
        out = self.sigmoid(np.matmul(hid_new,W2))
        return hid, out
```

In the forward pass function of the network, we have used the following functions.

$$J_1 = \frac{1}{2N}\sum_{i=1}^{N}\left\|X - \hat{X}\right\|^2$$

$$Y = activation\left(W_{out,all} * activation\left(W_{hidden,all} * \tilde{X}\right)\right)$$

$$where\ activation\ function\ is\ a\ sigmoid\ and\ W_{,all} = \begin{bmatrix} W \\ b \end{bmatrix}, \tilde{X} = \begin{bmatrix} X & 1 \end{bmatrix}$$

For aeCost, we need to find the derivatives of MSE, L2 regularization, and KL divergence.

$$MSE = J_1 = \frac{1}{2N}\sum_{i=1}^{N}\left\|X - \hat{X}\right\|^2$$

$$\delta_{out} = -\left(X - \hat{X}\right) \odot activation'(V_{out})$$

$$\delta_{hidden} = ((W_{out})^T \delta_{out}) \odot activation'(V_{hidden})$$

$$\nabla J_{1,Wout} = \delta_{out}(W_{out})^T$$

$$\nabla J_{1,bout} = \sum_i \delta_{out\,(i,j)}$$

$$\nabla J_{1,Whidden} = \delta_{hidden}(W_{hidden})^T$$

$$\nabla J_{1,bhidden} = \sum_i \delta_{hidden\,(i,j)}$$

$$L2\ Regularization = J_2 = \frac{\lambda}{2}\left[\sum_{b=1}^{Lhid}\sum_{a=1}^{Lin}(W_{a,b}^{(1)})^2 + \sum_{c=1}^{Lout}\sum_{b=1}^{Lhid}(W_{b,c}^{(2)})^2\right]$$

$$\nabla J_{2,Wout} = \lambda\, W^{(2)}$$
$$\nabla J_{2,Whidden} = \lambda\, W^{(1)}$$

$$KL\ Divergence = J_3 = \beta \sum_{b=1}^{Lhid} KL(\rho|\widehat{\rho_b}) = \beta \sum_{b=1}^{Lhid} \rho \log\left(\frac{\rho}{\widehat{\rho_b}}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\widehat{\rho_b}}\right)$$

$$where\ \widehat{\rho_b} = \frac{1}{N}\sum_{i=1}^{N} activation(W_{hidden,all} * \tilde{X})$$

$$\nabla J_{3,Wout} = 0$$

$$\nabla J_{3,Whidden} = \beta\left[-\left(\frac{\rho}{\widehat{\rho_b}}\right) + \left(\frac{1-\rho}{1-\widehat{\rho_b}}\right)\right]$$

```
    def update(self, We, data, params,l_rate):
        (Lin, Lhid, lmb, beta, rho) = params #Extra Parameters
        W1, W2, b1, b2 = We #Weights
        N = data.shape[0] #Sample Size
        J, Jgrad = self.aeCost(We, data, params)
        W1 = W1 - Jgrad[0]*l_rate
        W2 = W2 - Jgrad[1]*l_rate
        b1 = b1 - Jgrad[2]*l_rate
        b2 = b2 - Jgrad[3]*l_rate
        newWeights=(W1, W2, b1, b2)
        return J,newWeights

    def Train(self, We, data, params,l_rate, epochs, batch_size):
        for epoch in range(epochs):
            indexing=np.random.permutation(data.shape[0])
            data=data[indexing,:]
            numBatches = int(np.floor(data.shape[0]/batch_size))
            Total_Loss=0
            for j in range(numBatches):
                loss, We = self.update(We,data[j*batch_size:batch_size*(j+1),:],params,l_rate)
                Total_Loss=Total_Loss+loss

            Total_Loss=Total_Loss/numBatches
            print('Epoch', epoch+1)
            print('Loss', Total_Loss)
        return We
```

```
    def aeCost(self, We, data, params):
        (Lin, Lhid, lmb, beta, rho) = params #Extra Parameters
        W1, W2, b1, b2 = We #Weights
        N = data.shape[0] #Sample Size

        hidden, dataResult = self.fowardpass(We, data)
        hidden_mean = np.mean(hidden, axis=0)

        MSE = (1/(2*N))*np.sum(np.power((data - dataResult),2))
        Tyk = (lmb/2)*(np.sum(W1**2) + np.sum(W2**2))
        kl1 = rho*np.log(hidden_mean/rho)
        kl2 = (1-rho)*np.log((1-hidden_mean)/(1-rho))
        KL_final =  beta*np.sum(kl1+kl2)

        J = MSE + Tyk + KL_final

        deltaOut = -(data-dataResult)*self.sigmoidDerivative(dataResult)
        derKL = np.tile(beta*(-(rho/hidden_mean.T)+((1-rho)/(1-hidden_mean.T))), (N,1)).T
        deltaHid = (np.matmul(W2,deltaOut.T)+ derKL) * self.sigmoidDerivative(hidden).T

        gradWout = (1/N)*(np.matmul(deltaOut.T,hidden).T + lmb*W2)
        gradBout = np.mean(deltaOut, axis=0)
        gradWhid = (1/N)*(np.matmul(data.T,deltaHid.T) + lmb*W1)
        gradBhid = np.mean(deltaHid, axis=1)

        Jgrad=(gradWhid, gradWout, gradBhid, gradBout)

        return J, Jgrad
```

```
ae=Autoencoder()
We=ae.WeightInitialization(Linput,Lhid,Loutput)
We_update=ae.Train(We,Flat_Data,params,0.25,200,640)
```

```
Epoch 1
Loss 1.9826567970092597
Epoch 2
Loss 2.022958311125704
Epoch 3
Loss 2.0135698785251064
Epoch 4
Loss 2.00558096983518
Epoch 5
Loss 1.9969618014417168
Epoch 6
Loss 1.9869108146090935
Epoch 7
Loss 1.974940561428762
Epoch 8
Loss 1.960492203844587
```

Part C

This part asks us to retrieve the optimized weights and plot the hidden weights for each hidden layer neuron.

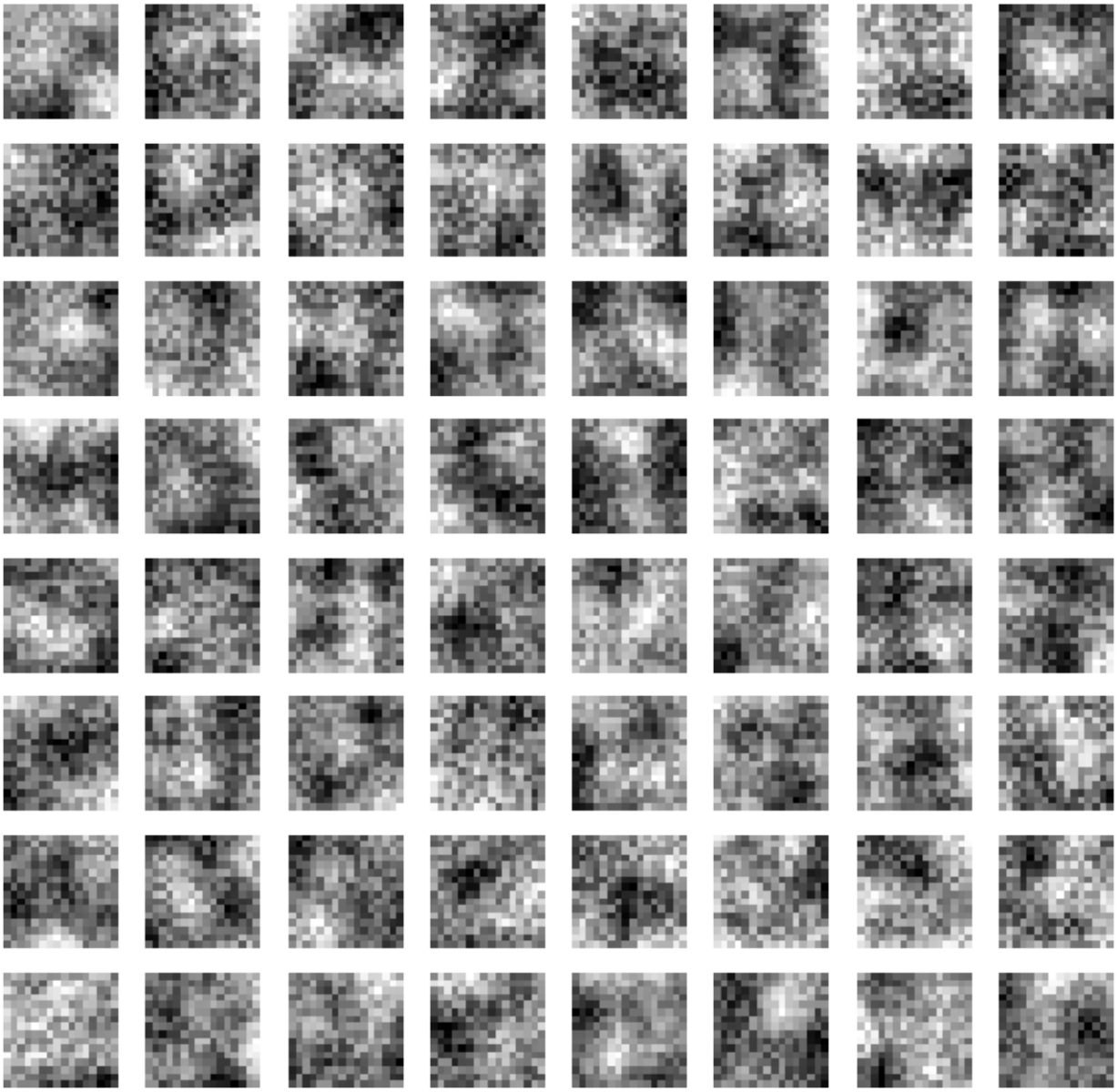


Figure 3. Visualizations of Hidden Layer Weights.

Here, we observe that the weight matrices resemble the natural images that are given in the dataset. Furthermore, we can say that extracted features also have learned various rotations in the data. The images resemble lines with depth in different orientations. Nevertheless, these images do not completely represent the natural image and some of the images resembles each other which might suggest that there are enough hidden layer units present to learn or some patterns are more abundant in the data than the others.

Part D

In this part, we are asked to implement the same network with different values of L_{hid} and λ. The available ranges for these values are as follows.

$$L_{hid} \in [10\ 100]$$

$$\lambda \in [0\ 10^{-3}]$$

The chosen values are L_{hid} = [12, 48, 96] and λ = [0, $5*10^{-4}$, 10^{-3}]. The observations from these images are as follows.

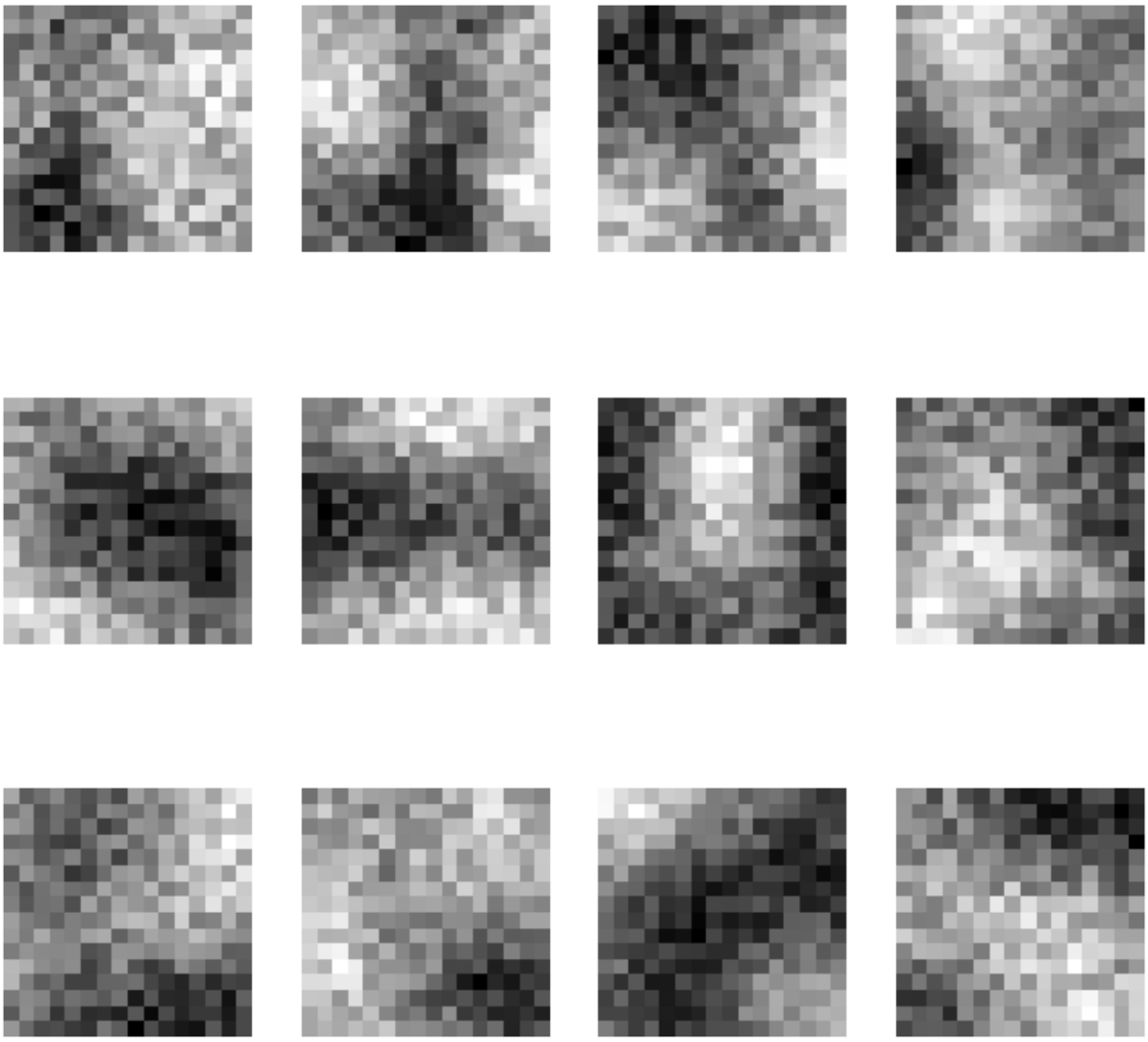


Figure 4. Visualizations of Hidden Layer Weights for 12 hidden neurons.

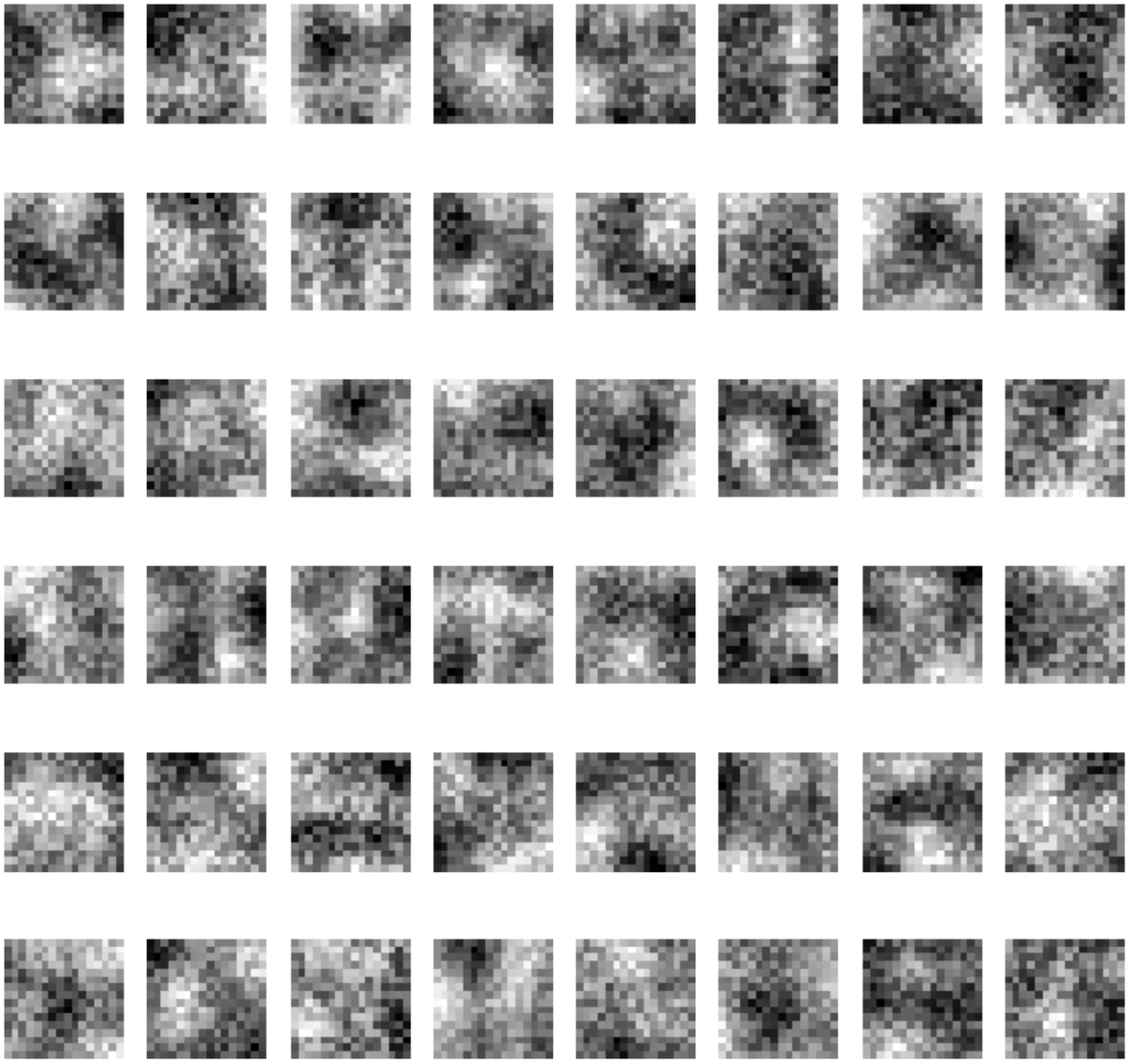


Figure 5. Visualizations of Hidden Layer Weights for 48 hidden neurons.

From figures 4, 5, and 6, we can see that number of neurons in the hidden layer affects the feature extraction quality of the autoencoder. We see that as the number of neurons increased, the number of complex patterns increased. In figure 4, we only see a crude representation of the patterns while as we increase the neuron number, these patterns become distinctive from each other and become more complex. However, if we have a neuron number after a threshold, it might lead to overfitting as there are only a certain number of important details to be learned. On the other hand, the network with 12 hidden layer units is incapable of learning the distribution at all since the capacity is not enough.

Figure 6. Visualizations of Hidden Layer Weights for 96 hidden neurons.

The following figures are acquired by changing λ while the neuron number is constant.

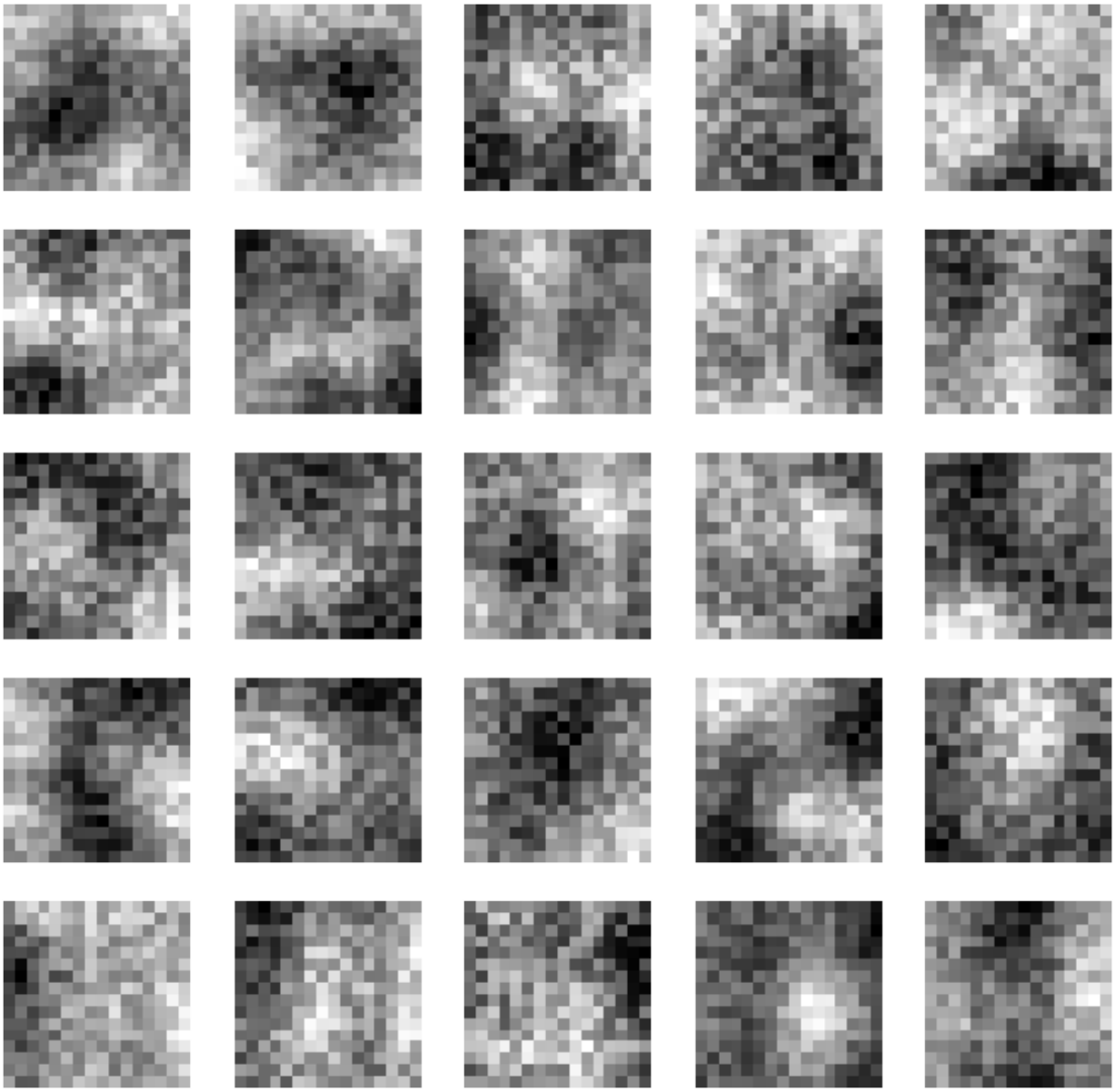


Figure 7. Visualizations of Hidden Layer Weights for λ=0.

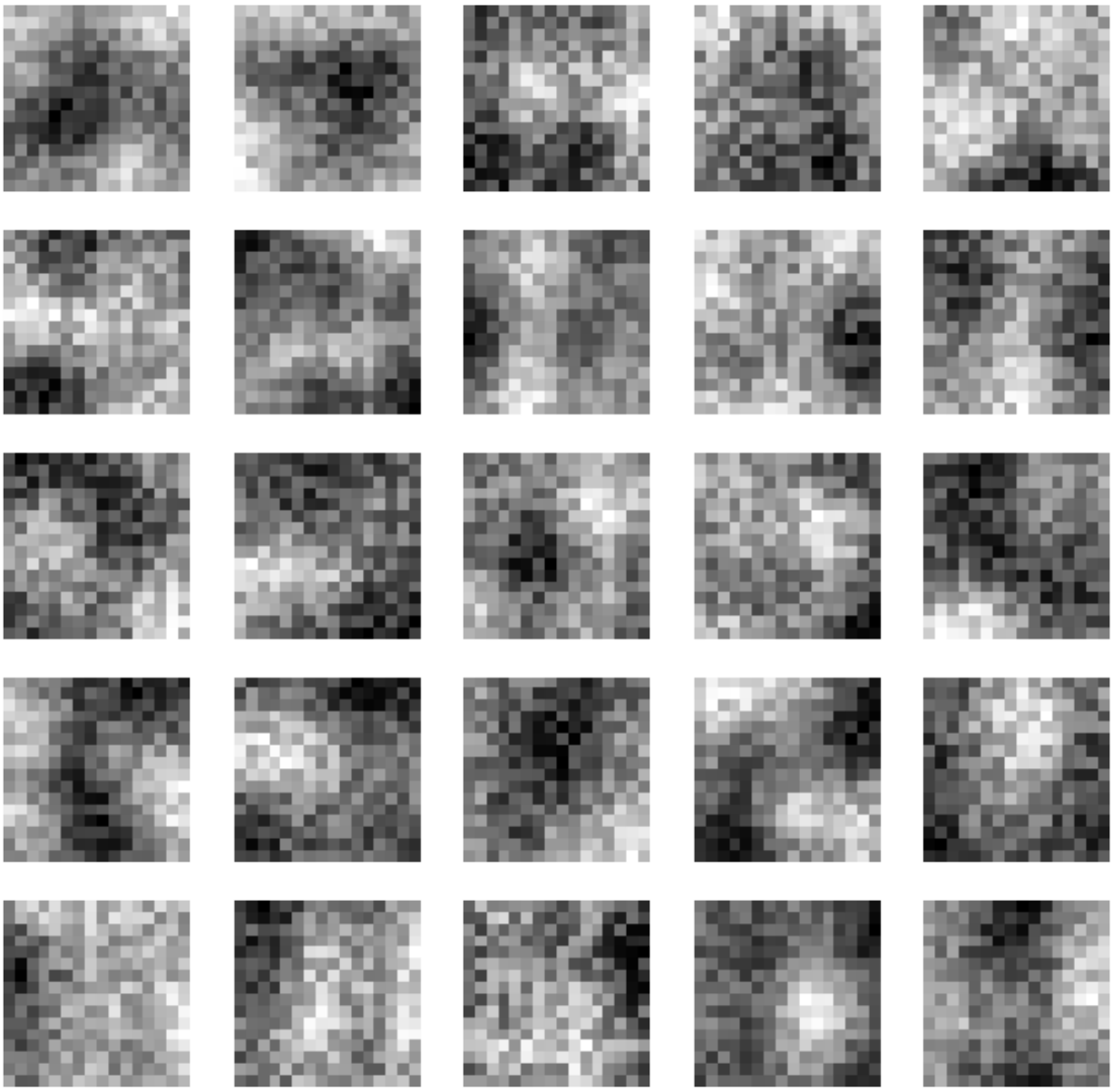


Figure 8. Visualizations of Hidden Layer Weights for λ=5e-4.

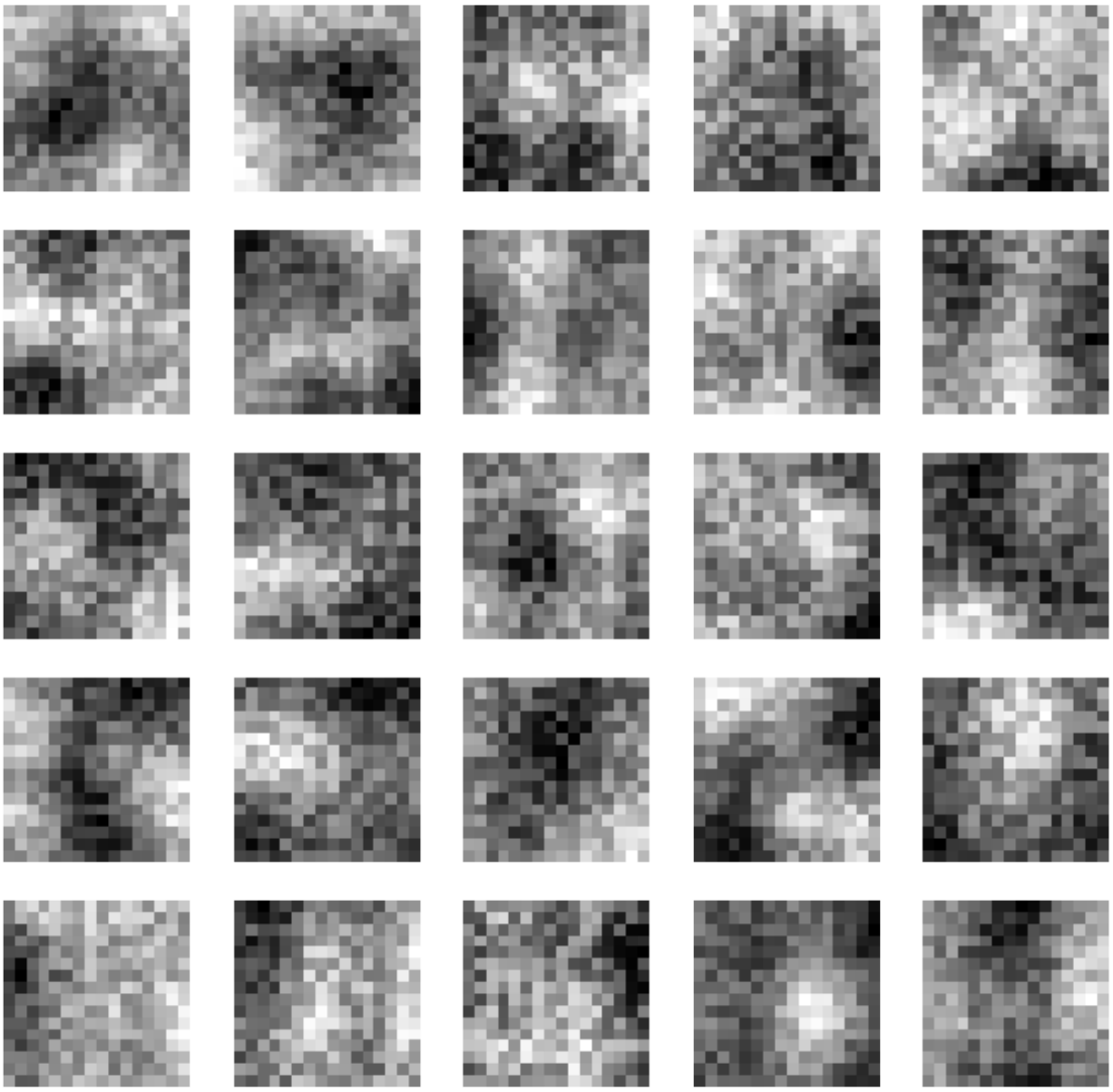


Figure 9. Visualizations of Hidden Layer Weights for λ=1e-3.

From figures 7-9, we can see that as we increase λ, images become smoother due to regularization. However, such effect is minimal and can only be checked by looking and the loss function. After running the autoencoder for different λ values, I have observed that as I have increased the λ, loss also increased. This was expected as λ helps us to regularize the autoencoder and avoid overfitting. By changing β and ρ values, their difference can be better observed. when looking at λ values, we see that as λ increases, the network becomes more generalized, however, we also see that after some point the regularization suppresses the loss and makes the network unable to learn.

Question 2

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 Convolutional Neural Networks and comment on the results. The notebook starts by importing the CIFAR-10 dataset which contains different classes of images. The CNN will be implemented using this dataset. In the first part, the naïve convolution layer has been implemented where the convolution operation takes place. The convolution operator works as an edge detection mechanism for the given kernels. Afterward, we implement a naïve backward pass that computes gradients. After the implementation of naïve backward pass, naïve max-pooling has been implemented. Pooling is used to summarize features, and by pooling the features become more generalized.

After the implementations of the naïve layer and pooling, we are provided with fast layers. It uses C-based Cython to speed up operations. Then, convolutional "sandwich" layers have been implemented to combine multiple operations into commonly used patterns. Three-layer ConvNet is defined afterward. With this convolutional network, we have applied sanity check the loss along with gradient check and overfitting with a small dataset. After checking, the training is done with one epoch and the validation accuracy becomes around 50%. Next, we have implemented a special batch normalization layer with its forward and backward pass where it accepts inputs of shape (N, C, H, W) and produce outputs of shape (N, C, H, W) where the N dimension gives the minibatch size and the (H, W) dimensions give the spatial size of the feature map. For the convolution case, each kernel is normalized in itself around all of the batches. Lastly, we have implemented group normalization. The code along with the outputs is given at the end of the report.

Part B

This notebook serves as an introductory chapter for PyTorch or Tensorflow. I have chosen PyTorch to work with as we are planning to utilize PyTorch in our project. PyTorch uses an array model called tensor which is very similar to a NumPy array. PyTorch also enables us to harness the power of our GPU. Thus, we have an option to utilize CUDA. In order to increase the speed of calculations, I have used my GPU for this part. The notebook consists of five parts. Loading the CIFAR-10 dataset, using Barebone PyTorch, PyTorch Model API, PyTorch Sequential API, and an Open-Ended Challenge. After loading the dataset and dividing it to test, train, and validation the dataset, we start with the flattening function. Tensor is of shape N x C x H x W, where N means the number of data points, C means channel number, H means height and W means width data. To make it analogous to NumPy, we implement reshape function with flatten(). Then, two_layer_fc has been implemented for a two-layer network with ReLU activation function. The input is flattened and the weight parameters are initialized with the given parameters. As the notebook states, we don’t need to store gradient values or the backpropagation process since PyTorch handles intermediate values.

Afterward, we implement a three-layer convolutional neural network that contains “sandwiched” layers that we have seen in the CNN notebook. The notebook gives us two different initialization functions. One is for creating random weights while the other one is for creating zero weights. Furthermore, a training loop along with the check accuracy has been implemented. After the creation of BareBones PyTorch functions, we have switched to PyTorch module API. It seems that this is the most effective way of utilizing PyTorch capabilities as it provides flexibility with ease of use. The Sequential model is created by stacking network layers and specifying an optimizer. A two-layer network and the three-layer convolutional neural network has been also implemented using PyTorch module API and its performance exceeds the barebones implementation. Lastly, PyTorch Sequential API has been implemented which is not flexible but very convenient and easy to implement. At the final challenge, I have added an affine layer and played with the learning rate, added weight decay, implemented softmax, and add ReLu function to the affine layer. Nonetheless, I have achieved the best result when I have only implemented a single layer affine network with an increased learning rate. The final test score was 74.49% which is an improvement.

Describe what you did

In the cell below you should write an explanation of what you did, any additional features that you implemented, and/or any graphs that you made in the process of training and evaluating your network.

TODO: I managed to increase the validation accuracy to 74.5% and test accuracy to 74.49%. To do this I did not touch the convolutional layers and simply extended the fully connected layers, that is named layer 4. I used a L2 regularization with the weight decay of 1e-6 to decrease the chance of overfitting. However, this affected my accuracy badly. Thus, I have removed it and increased the learning rate instead. I used a ReLU layer to increase complex learning. Furthermore, in the second layer of layer 4, I have also used Softmax as an non-linear activation function. Nonetheless, I did not achieve high precision with neither ReLu nor Softmax. Thus, I have erased them and only used affine layer. It was great practice that will prove beneficial in the final project.

Test set -- run this only once

Now that we've gotten a result we're happy with, we test our final model on the test set (which you should store in best_model). Think about how this compares to your validation set accuracy.

```
best_model = model
check_accuracy_part34(loader_test, best_model)
```

```
Checking accuracy on test set
Got 7449 / 10000 correct (74.49)
```

The codes along with the output results are in the final pages of the report.

Question 3

In this question, we are asked to implement a network to classify human activity by using incoming data from three different sensors over T = 150-time samples. The training set consists of 3000 samples, and the test set consists of 600 samples. I have first loaded the dataset.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import h5py
import seaborn as sn
filename_Q3 = "assign3_data3.h5"
```

```
file = h5py.File(filename_Q3, "r") #Load Data
```

```
# List all groups
print("Keys: %s" % list(file.keys()))
trX = list(file.keys())[0]
trY = list(file.keys())[1]
tstX = list(file.keys())[2]
tstY = list(file.keys())[3]

train_data = np.array(file[trX])
train_labels = np.array(file[trY])
test_data = np.array(file[tstX])
test_labels = np.array(file[tstY])
```

```
Keys: ['trX', 'trY', 'tstX', 'tstY']
```

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

print("Train Data Size & Train Label Size:", train_data.shape,train_labels.shape)
#The length of each time series is 150 units.
#Labels: (downstairs=1, jogging=2, sitting=3, standing=4, upstairs=5, walking=6)
```

```
Number of Train Samples: 3000
Train Data Size & Train Label Size: (3000, 150, 3) (3000, 6)
```

Part A

In this part, we are asked to implement a single-layer RNN architecture with 128 hidden neuron numbers along with hyperbolic tangent activation to classify human activity. For this part, I have used a stochastic gradient descent algorithm, mini-batch size of 32 samples, the learning rate of η = 0.1, momentum rate of = 0.85, maximum of 50 epochs. For weight initialization, we are asked to implement Xavier Uniform distribution. Its equation is as follows.

$$r = \sqrt{\frac{6}{L_{prev} + L_{next}}}, W = [-r, r]$$

For equal comparison, the hyperparameters are kept the same throughout Part A, Part B, and Part C. Forward propagation and backpropagation algorithms have been defined in Assignment 2. Thus, I have used the same algorithm in this assignment as well. I have first defined the layer class.

```
#Build Class for RNN and MLP

class Layer: #Layer Class
    def __init__(self,inputDim,numNeurons,activation,beta):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation
        self.beta=beta
        self.w0=np.sqrt(6/(inputDim+numNeurons))
        self.W1=np.random.uniform(-self.w0,self.w0,(inputDim,numNeurons))
        self.b1=np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.weightsAll = np.concatenate((self.W1, self.b1), axis=0)
        self.W2=np.random.uniform(-self.w0,self.w0,(numNeurons,numNeurons))

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

    def activationFunction(self, x):
        if(self.activation == 'hyperbolic'):
            return np.tanh(self.beta*x)
        elif(self.activation == 'softmax'):
            exp_x = np.exp(x - np.max(x))
            return exp_x/np.sum(exp_x, axis=1 ,keepdims=True)
        elif(self.activation=="relu"):
            return np.maximum(x,0)
        elif(self.activation=="sigmoid"):
            exp_x = np.exp(2*x)
            return exp_x/(1+exp_x)
        else:
            return x
    def activationNeuron(self,x):
        x=np.array(x)
        numSamples = x.shape[0]
        tempInp = np.concatenate((x, -1*np.ones((numSamples, 1))), axis=1)
        self.lastActiv = self.activationFunction(np.matmul(tempInp,self.weightsAll))
        return self.lastActiv

    def RecurrentActivation(self,x,hid):
        x=np.array(x)
        numSamples = x.shape[0]
        tempInp = np.concatenate((x, -1*np.ones((numSamples, 1))), axis=1)
        final=np.matmul(tempInp,self.weightsAll)+np.matmul(hid,self.W2)
        self.lastActiv = self.activationFunction(final)
        return self.lastActiv

    def activation_derivative(self, x):
        if(self.activation == 'hyperbolic'):
            return self.beta*(1-(x**2))
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation=="sigmoid"):
            return (x*(1-x))
        elif (self.activation=="relu"):
            return 1*(x>0)
        else:
            return np.ones(x.shape)
```

Notice that I have defined a placeholder function in the activation derivative function for softmax. However, this value will not be used and it is there for placeholder purposes. I have defined the necessary functions for each class and I have also implemented “RecurrentActivation” function. In this function, we use both the input value and the previous state value is taken as inputs. After creating the layer class, I have implemented RNN_Classifier class where the whole network is fully realized. Using the layer function, the mathematical interpretation of the RNN at each time step t for our problem is as follows.

$$h(t) = \tanh(x(t) * W_{ih} + h(t-1) * W_{hh} + b)$$

where i: input neuron number, h: hidden layer neuron number.

```
class RNN_Classifier:
    def __init__(self,training_inputs):
        self.layers=[]
        self.NumSample, self.TimeSample, self.D = training_inputs.shape
        self.Hidden_prev=np.zeros((32, 128))
        self.RecurrentError=np.empty((32, self.TimeSample, 128))
        self.RecurrentDelta=np.empty((32, self.TimeSample, 128))

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

    def FowardProp(self,training_inputs):
        #Foward Propagation

        #RNN Layer - First Layer
        NumSample, TimeSample, D = training_inputs.shape
        IN=np.empty((NumSample, TimeSample, 128))
        self.Hidden_prev=np.zeros((NumSample, 128))
        for time in range(TimeSample): #You have to take into account whole time samples.
            x = training_inputs[:, time, :]
            IN[:, time, :]=self.layers[0].RecurrentActivation(x,self.Hidden_prev)
            self.Hidden_prev=IN[:, time, :]
        OUT = IN[:, -1, :]

        for layer in self.layers[1:len(self.layers)]: #For MLP Layers
            OUT=layer.activationNeuron(OUT) #Only use the last time sample as it contains memory.
        return IN,OUT
```

In the Forward propagation function, I have separated the forward propagation of the recurrent layer with MLP layers. In the recurrent layer, we start from time sample 0 and move towards the time of the end. At each iteration, we activate the RecurrentActivation function and assign the current input value to be the next previous state value. MLP forward propagation is pretty straightforward.

```
    def BackProp(self,l_rate,batch_size,training_inputs,training_labels,momentCoef):
        IN,OUT = self.FowardProp(training_inputs)
        foward_out = OUT
        for i in reversed(range(len(self.layers))):# Backpropagation until recurrent
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.lyrDelta=training_labels-foward_out
            elif (lyr == self.layers[0]):
                nextLyr = self.layers[i+1]
                lyr.lyrError = np.matmul(nextLyr.lyrDelta, nextLyr.weightsAll[0:nextLyr.weightsAll.shape[0]-1,:].T)
                self.RecurrentError[:,-1,:]=lyr.lyrError
                derivative=lyr.activation_derivative(lyr.lastActiv)
                lyr.lyrDelta=derivative*lyr.lyrError
                self.RecurrentDelta[:,-1,:]=lyr.lyrDelta
            else:
                nextLyr = self.layers[i+1]
                lyr.lyrError = np.matmul(nextLyr.lyrDelta, nextLyr.weightsAll[0:nextLyr.weightsAll.shape[0]-1,:].T)
                derivative=lyr.activation_derivative(lyr.lastActiv)
                lyr.lyrDelta=derivative*lyr.lyrError
        dWall=0
        dWhid=0
        NumSample, TimeSample, D = training_inputs.shape
        Prev=np.empty((NumSample, TimeSample, 128))
        #Backpropagation through time (Recurrent)
        for time in reversed(range(TimeSample)):
            lyr=self.layers[0]
            #u = IN[:, time-1, :]
            x=training_inputs[:,time,:]

            if time > 0:
                u = IN[:, time-1, :]
            else:
                u = np.zeros((NumSample, 128))
```

```
            derivative=lyr.activation_derivative(u)
            dWhid+=np.matmul(u.T, self.RecurrentDelta[:,time,:])
            dWall+=np.matmul(np.concatenate((training_inputs[:,time-1,:], -1*np.ones((training_inputs[:,time-1,:].shape

            #For Recurrent Delta Update
            self.RecurrentError[:,time-1,:]=np.matmul(self.RecurrentDelta[:,time,:],lyr.W2.T)
            self.RecurrentDelta[:,time-1,:]=derivative*self.RecurrentError[:,time-1,:]

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                update_1 =  l_rate*dWall/(batch_size*150)
                update_2 =  l_rate*dWhid/(batch_size*150)
                lyr.weightsAll+= update_1 + (momentCoef*lyr.prevUpdate)
                lyr.W2+=update_2 + (momentCoef*lyr.prevUpdateRNN)
                lyr.prevUpdate = update_1
                lyr.prevUpdateRNN = update_2

            else:
                numSamples = self.layers[i - 1].lastActiv.shape[0]
                tempInp=np.concatenate((self.layers[i - 1].lastActiv, -1*np.ones((numSamples, 1))), axis=1)
                update =  l_rate*np.matmul(tempInp.T, lyr.lyrDelta)/batch_size
                lyr.weightsAll+= update + (momentCoef*lyr.prevUpdate)
                lyr.prevUpdate = update
```

The backpropagation algorithm of assignment 2 has been kept the same for MLP layers. After the last MLP layer, our delta goes through the backpropagation through time (BPTT) algorithm. During BPTT, delta has been computed similar to MLP backpropagation. However, this time, we iterate delta up to time 0. For each iteration, we update dWhid and dWall which represent the matrix multiplication of input values with delta. After the iteration ends, weights are updated for each layer.

```
    def TrainNetwork(self,l_rate,batch_size,training_inputs,training_labels, test_inputs, test_labels, epochNum,momentC
        crossList = []
        TrainList=[]
        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 = training_inputs[j*batch_size:batch_size*(j+1),:,:]
                train_labels = training_labels[j*batch_size:batch_size*(j+1),:]
                self.BackProp(l_rate,batch_size,train_data,train_labels,momentCoef)
            IN, valOutput = self.FowardProp(test_inputs)
            IN1, TrainOutput = self.FowardProp(training_inputs)
            crossErr =  np.sum(-np.log(valOutput) * test_labels)/valOutput.shape[0]
            crossErr1 =  np.sum(-np.log(TrainOutput) * training_labels)/TrainOutput.shape[0]
            print('Cross-Entropy Error for Validation', crossErr)
            print('Cross-Entropy Error for Train', crossErr1)
            crossList.append(crossErr)
            TrainList.append(crossErr1)
        return crossList, TrainList

    def Predict(self,inputs,output_real):
        Output = self.FowardProp(inputs)[1]
        Output = Output.argmax(axis=1)
        output_real = output_real.argmax(axis=1)
        return ((Output == output_real).mean()*100)
```

TrainNetwork function has been created in order to encapsulate both forward propagations along with backpropagation and BPTT. Furthermore, it enables us to mix the samples and create batches to be used for SGD with mini-batch.

```
    def ConfusionMatrix(self,input1,output1):
        prediction= self.FowardProp(input1)[1]
        prediction = prediction.argmax(axis=1)
        output1 = output1.argmax(axis=1)
        K = len(np.unique(output1))
        c=np.zeros((K,K))
        for i in range(len(output1)):
            c[output1[i]][prediction[i]] += 1
        return c
```

After creating the classes, I have inputted the hyperparameters.

```
momentum = 0.85
l_rate = 0.1
epoch = 30
batch_size = 32
RNN_Neuron=128
indexing=np.random.permutation(train_data.shape[0])
train_data=train_data[indexing,:,:]
train_labels=train_labels[indexing,:]

val_size = int(train_data.shape[0] / 10)
val_data=train_data[:val_size,:,:]
val_labels=train_labels[:val_size,:]
train_data1=train_data[val_size:,:,:]
train_labels1=train_labels[val_size:,:]

RNN_Net = RNN_Classifier(train_data1)
RNN_Net.addLayer(Layer(3, RNN_Neuron,'hyperbolic',1))
RNN_Net.addLayer(Layer(RNN_Neuron,70,'relu',1))
RNN_Net.addLayer(Layer(70,30,'relu',1))
RNN_Net.addLayer(Layer(30,6,'softmax',1))
crossList, TrainList = RNN_Net.TrainNetwork(l_rate,batch_size,train_data1,train_labels1,val_data,val_labels,epoch,moment
```

I have used ReLu for MLP layers as it provides higher performance compared to sigmoid or tanh functions.

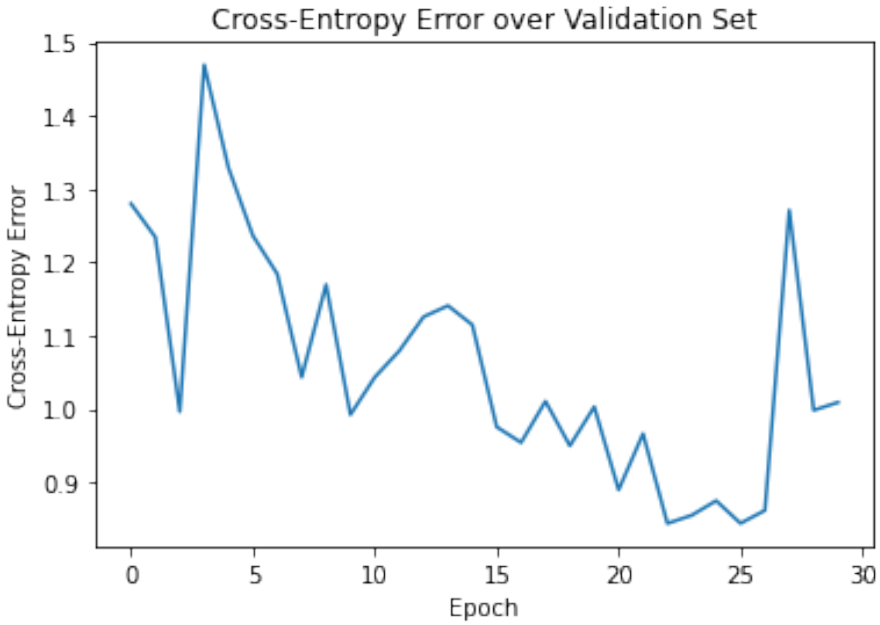


Figure 10. Cross entropy of validation dataset for RNN.

```
Test Accuracy: 49.5%
Train Accuracy: 59.851851851851855%
```

After the training process, our Test accuracy is around 50%. This is not a great classifier. Furthermore, we can see from figure 10 that it has fluctuations which means the network is not stable. During the training process, I have experimented with different learning rates and I have concluded that if I increase the learning rate, the loss explodes and yields NaN. This happens due to the cumulative nature of BPTT. After each iteration, our gradient accumulates higher and higher values. This is the problem of RNN architectures as they suffer from vanishing/exploding gradients.

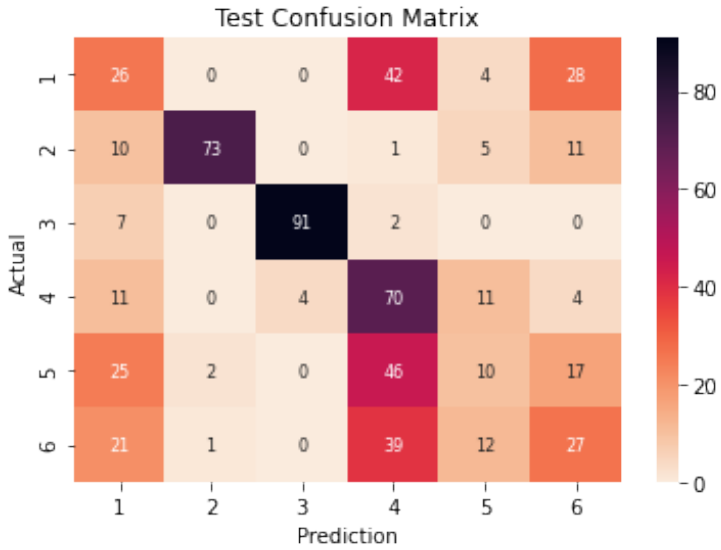


Figure 11. Confusion matrix of test dataset.

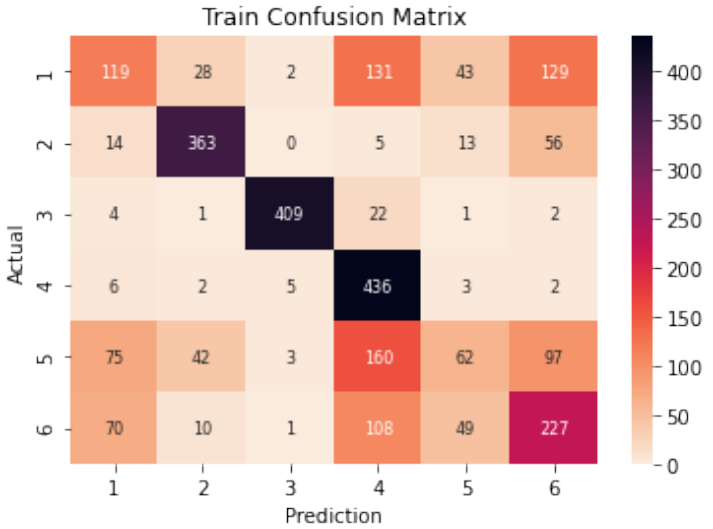


Figure 12. Confusion matrix of train dataset.

Furthermore, we can see from figures 11 and 12 that the network only learns a portion of the data, hence cannot predict most of the classes correctly.

Part B

In this part, we are asked to again implement a classifier. However, this time, we are asked to implement LSTM instead of RNN. LSTM is created to solve (or reduce) the issues that arise from recurrent layers such as the vanishing/exploding gradient problem.

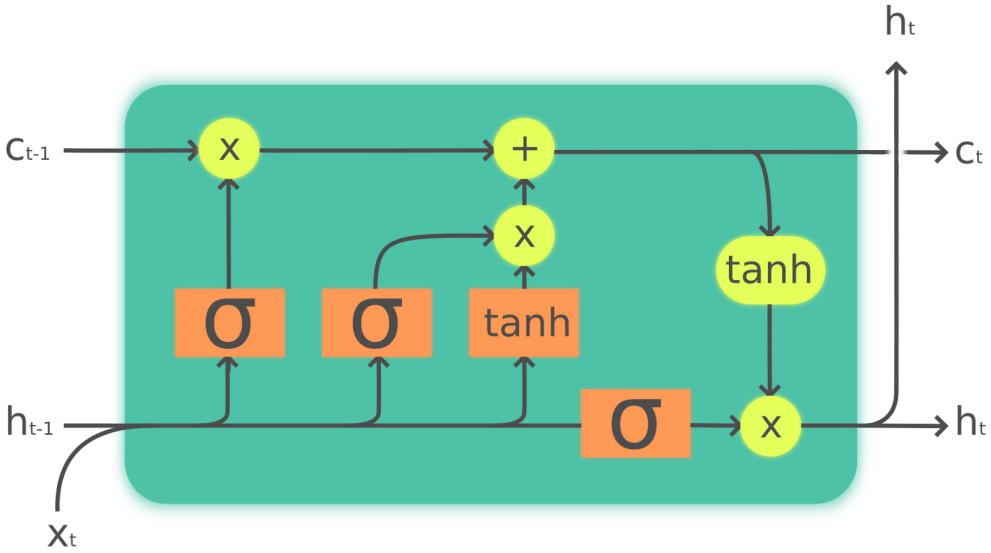


Figure 13. LSTM cell.

Forward propagation equations are as follows.

$$hf(t) = \sigma([h(t-1), x(t)] * W_f + b_f)$$

$$hi(t) = \sigma([h(t-1), x(t)] * W_i + b_i)$$

$$hc(t) = tanh([h(t-1), x(t)] * W_c + b_c)$$

$$ho(t) = \sigma([h(t-1), x(t)] * W_o + b_o)$$

$$c(t) = hf(t) * c(t-1) + hi(t) * hc(t)$$

$$h(t) = ho(t) * \tanh(c(t))$$

For the backpropagation through time algorithm, we can look at figure 14.

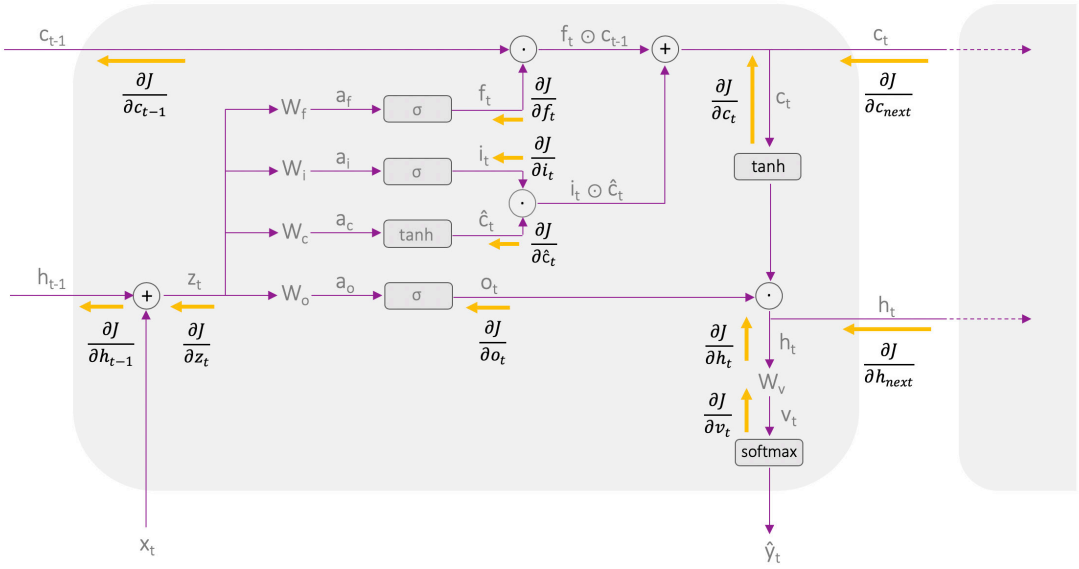


Figure 14. BPTT for LSTM cell.

I have used the same layer class that was present in part A for MLP layers. However, I have created an extra LSTM_Layer class for specifying LSTM variables.

```
#Build Class for LSTM
class LSTM_Layer: #LSTM Layer Class
    def __init__(self,inputDim,numNeurons,beta):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.beta=beta
        self.w0=np.sqrt(6/(inputDim+numNeurons))
        # forget gate
        self.W_f=np.random.uniform(-self.w0,self.w0,(inputDim,numNeurons))
        self.bf=np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.Wf = np.concatenate((self.W_f, self.bf), axis=0)
        # input gate
        self.W_i=np.random.uniform(-self.w0,self.w0,(inputDim,numNeurons))
        self.bi=np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.Wi = np.concatenate((self.W_i, self.bi), axis=0)
        # cell gate
        self.W_c=np.random.uniform(-self.w0,self.w0,(inputDim,numNeurons))
        self.bc=np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.Wc = np.concatenate((self.W_c, self.bc), axis=0)
        # output gate
        self.W_o=np.random.uniform(-self.w0,self.w0,(inputDim,numNeurons))
        self.bo=np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.Wo = np.concatenate((self.W_o, self.bo), axis=0)
```

```
        self.lastActiv=None
        self.lyrDelta=None
        self.lyrError=None
        self.prevUpdate_f = 0
        self.prevUpdate_i = 0
        self.prevUpdate_c = 0
        self.prevUpdate_o = 0
```

```
    def activationFunction(self, x, activation):
        if(activation == 'hyperbolic'):
            return np.tanh(self.beta*x)
        elif(activation == 'softmax'):
            exp_x = np.exp(x - np.max(x))
            return exp_x/np.sum(exp_x, axis=1)
        elif(activation=="relu"):
            return np.maximum(x,0)
        elif(activation=="sigmoid"):
            exp_x = np.exp(2*x)
            return exp_x/(1+exp_x)
        else:
            return x
    def activationNeuron(self,x, w, activation):
        x=np.array(x)
        numSamples = x.shape[0]
        tempInp = np.concatenate((x, -1*np.ones((numSamples, 1))), axis=1)
        self.lastActiv = self.activationFunction(np.matmul(tempInp,w),activation)
        return self.lastActiv

    def RecurrentActivation(self,x,hid, activation):
        x=np.array(x)
        numSamples = x.shape[0]
        tempInp = np.concatenate((x, -1*np.ones((numSamples, 1))), axis=1)
        final=np.matmul(tempInp,self.weightsAll)+np.matmul(hid,self.W2)
        self.lastActiv = self.activationFunction(final,activation)
        return self.lastActiv

    def activation_derivative(self, x ,activation):
        if(activation == 'hyperbolic'):
            return self.beta*(1-(x**2))
        elif(activation == 'softmax'):
            return x*(1-x)
        elif(activation=="sigmoid"):
            return (x*(1-x))
        elif (activation=="relu"):
            return 1*(x>0)
        else:
            return np.ones(x.shape)
```

The only difference here is that I have initialized extra weights along with some other variables. LSTM_Classifier class is again similar to the RNN case. Thus, I have only added here the changed parts.

After the implementation of classes, I have used the same parameters for LSTM_Classifier training as well.

```
def FowardProp(self,training_inputs): #CHECKED
    #Foward Propagation
    #LSTM Layer - First Layer
    lyr = self.layers[0]
    N, T, D = training_inputs.shape
    H=128

    z = np.empty((N, T, D + H))
    c = np.empty((N, T, H))
    tanhc = np.empty((N, T, H))
    hf = np.empty((N, T, H))
    hi = np.empty((N, T, H))
    hc = np.empty((N, T, H))
    ho = np.empty((N, T, H))

    h_prev=np.zeros((N, H))
    c_prev=np.zeros((N, H))
    #Though it looks complex, just applying the functions
    for t in range(T):
        z[:, t, :] = np.column_stack((h_prev, training_inputs[:, t, :]))
        zt = z[:, t, :]
        hf[:, t, :] = lyr.activationNeuron(zt , lyr.Wf, "sigmoid")
        hi[:, t, :] = lyr.activationNeuron(zt, lyr.Wi, "sigmoid")
        hc[:, t, :] = lyr.activationNeuron(zt, lyr.Wc, "hyperbolic")
        ho[:, t, :] = lyr.activationNeuron(zt, lyr.Wo, "sigmoid")

        c[:, t, :] = hf[:, t, :] * c_prev + hi[:, t, :] * hc[:, t, :]

        tanhc[:, t, :] = lyr.activationFunction(c[:, t, :], "hyperbolic")

        h_prev = ho[:, t, :] * tanhc[:, t, :]
        c_prev = c[:, t, :]

        cache = {"z_summ": z, #Summation of h_t-1 and x_t
                 "c": c,  #Memory C_t
                 "tanhc": (tanhc), #Tanh of Memory C_t
                 "hf": hf, #Output h_f
                 "hi": (hi), #Output h_i
                 "hc": (hc),#Output h_c
                 "ho": (ho)}#Output h_o


    for layer in self.layers[1:len(self.layers)]: #For MLP Layers
        h_prev=layer.activationNeuron(h_prev)
    OUT= h_prev
    return cache,OUT
```

```
def BackProp(self,l_rate,batch_size,training_inputs,training_labels,momentCoef):
    cache,OUT = self.FowardProp(training_inputs)
    foward_out = OUT
    z = cache["z_summ"]
    c=cache["c"]
    tanhc=cache["tanhc"]
    hf=cache["hf"]
    hi=cache["hi"]
    hc=cache["hc"]
    ho=cache["ho"]

    for i in reversed(range(len(self.layers))):# Backpropagation until LSTM
        lyr = self.layers[i]
        #outputLayer
        if(lyr == self.layers[-1]):
            lyr.lyrDelta=training_labels-foward_out
        elif(lyr==self.layers[0]):
            nextLyr = self.layers[i+1]
            lyr.lyrError = np.matmul(nextLyr.lyrDelta, nextLyr.weightsAll[0:nextLyr.weightsAll.shape[0]-1,:].T)
            lyr.lyrDelta=lyr.lyrError

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

    dWf = 0
    dWi = 0
    dWc = 0
    dWo = 0
    H=128
    T = z.shape[1]
    NumSample, TimeSample, D = training_inputs.shape
    Prev=np.empty((NumSample, TimeSample, 128))
```

```
lyr0=self.layers[0]
delta=lyr0.lyrDelta
#Backpropagation through time (LSTM)
for t in reversed(range(T)):
    u = z[:, t, :]
    # if t = 0, c = 0
    if t > 0:
        c_prev = c[:, t - 1, :]
    else:
        c_prev = 0

    dc = delta * ho[:, t, :] * lyr0.activation_derivative(tanhc[:, t, :],"hyperbolic")
    dhf = dc * c_prev * lyr0.activation_derivative(hf[:, t, :],"sigmoid")
    dhi = dc * hc[:, t, :] * lyr0.activation_derivative(hi[:, t, :],"sigmoid")
    dhc = dc * hi[:, t, :] * lyr0.activation_derivative(hc[:, t, :],"sigmoid")
    dho = delta * tanhc[:, t, :] * lyr0.activation_derivative(ho[:, t, :],"sigmoid")

    dWf += np.matmul(np.concatenate((u, -1*np.ones((NumSample, 1))), axis=1).T, dhf)
    dWi += np.matmul(np.concatenate((u, -1*np.ones((NumSample, 1))), axis=1).T, dhi)
    dWc += np.matmul(np.concatenate((u, -1*np.ones((NumSample, 1))), axis=1).T, dhc)
    dWo += np.matmul(np.concatenate((u, -1*np.ones((NumSample, 1))), axis=1).T, dho)

    # update the error gradient.
    dxf = np.matmul(dhf, lyr0.Wf.T[:, :H])
    dxi = np.matmul(dhi, lyr0.Wi.T[:, :H])
    dxc = np.matmul(dhc, lyr0.Wc.T[:, :H])
    dxo = np.matmul(dho, lyr0.Wo.T[:, :H])

    delta = (dxf + dxi + dxc + dxo)

#Updates Weights for first and mlp layers
for i in range(len(self.layers)):
    lyr = self.layers[i]
    if(i == 0):

        update_f =  l_rate*dWf/(batch_size)
        update_i =  l_rate*dWi/(batch_size)
        update_c =  l_rate*dWc/(batch_size)
        update_o =  l_rate*dWo/(batch_size)

        lyr.Wf+= update_f + (momentCoef*lyr.prevUpdate_f)
        lyr.Wi+= update_i + (momentCoef*lyr.prevUpdate_i)
        lyr.Wc+= update_c + (momentCoef*lyr.prevUpdate_c)
        lyr.Wo+= update_o + (momentCoef*lyr.prevUpdate_o)

        lyr.prevUpdate_f = update_f
        lyr.prevUpdate_i = update_i
        lyr.prevUpdate_c = update_c
        lyr.prevUpdate_o = update_o
```

```
    delta = (dxf + dxi + dxc + dxo)

#Updates Weights for first and mlp layers
for i in range(len(self.layers)):
    lyr = self.layers[i]
    if(i == 0):

        update_f =  l_rate*dWf/(batch_size)
        update_i =  l_rate*dWi/(batch_size)
        update_c =  l_rate*dWc/(batch_size)
        update_o =  l_rate*dWo/(batch_size)

        lyr.Wf+= update_f + (momentCoef*lyr.prevUpdate_f)
        lyr.Wi+= update_i + (momentCoef*lyr.prevUpdate_i)
        lyr.Wc+= update_c + (momentCoef*lyr.prevUpdate_c)
        lyr.Wo+= update_o + (momentCoef*lyr.prevUpdate_o)

        lyr.prevUpdate_f = update_f
        lyr.prevUpdate_i = update_i
        lyr.prevUpdate_c = update_c
        lyr.prevUpdate_o = update_o

    else:
        numSamples = self.layers[i - 1].lastActiv.shape[0]
        tempInp=np.concatenate((self.layers[i - 1].lastActiv, -1*np.ones((numSamples, 1))), axis=1)
        update =  l_rate*np.matmul(tempInp.T, lyr.lyrDelta)/batch_size
        lyr.weightsAll+= update + (momentCoef*lyr.prevUpdate)
        lyr.prevUpdate = update
```

After the training process, we see a more stable convergence compared to the RNN case. Nonetheless, there are still spikes which represents some instability in the system. The cross-entropy plot can be seen in figure 15.

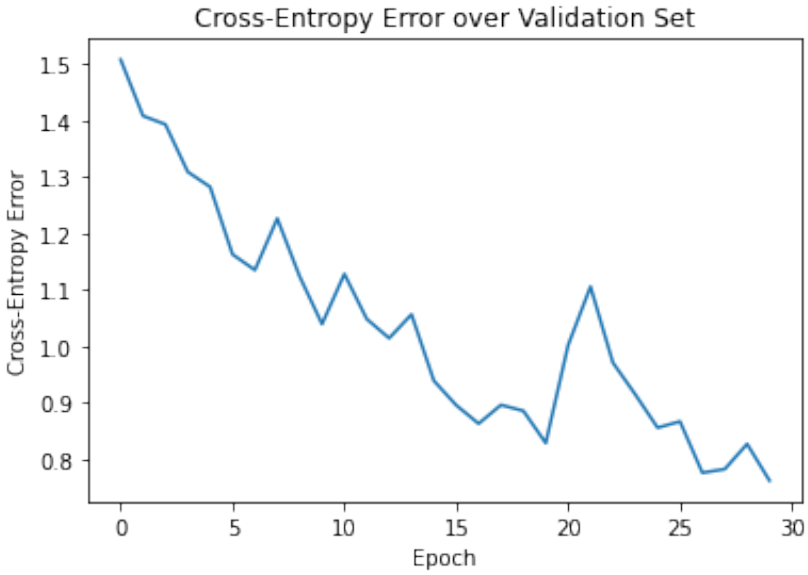


Figure 15. Cross entropy of validation dataset for LSTM.

After the training, test accuracy turns out to be 58% while the training accuracy is 68.4%.

```
Test Accuracy: 57.999999999999996%
Train Accuracy: 68.37037037037037%
```

This is an improvement over the RNN case. Nonetheless, it seems that there is still room for improvement. In the end, we can conclude that LSTM minimizes the problem of vanishing gradients/exploding gradients, and is much less susceptible to changes in the gradient updates. Confusion matrices also confirm this as they show that the network is able to learn the class patterns much better in its current state.

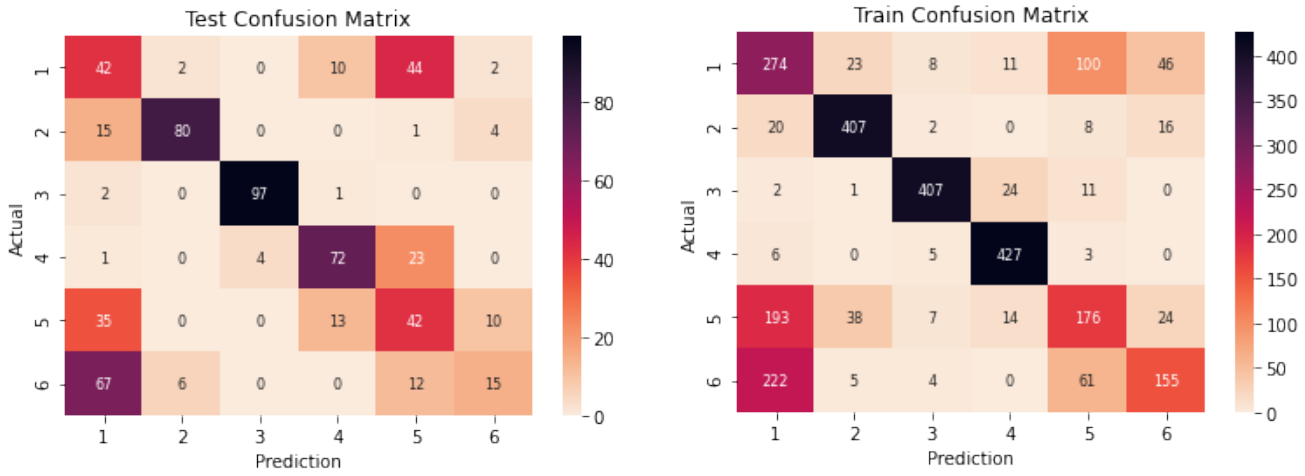
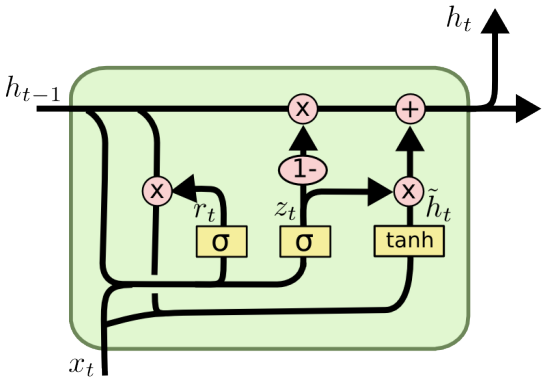


Figure 16,17. Confusion matrix of test dataset and train dataset.

Part C

In this part, we are asked to implement GRU which is expected to increase the accuracy of the network even higher. GRU is similar to LSTM structure-wise with a lesser gate number. The forward propagation algorithm along with the cell structure can be seen in figure 18.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 18. GRU cell structure along with forward propagation algorithm.

Due to the lesser gate size of the GRU, it takes less time to train the GRU network given that other parameters are held constant. Furthermore, GRU is much more memory efficient since it doesn't require the additional gate that LSTM includes.

For creating the classes for the GRU_Classifier, we can say that the GRU layer class is similar to LSTM with different initialized weights. Thus, to prevent this report from being cluttered, I will not put the pictures of the code here but at the end of the report as its structure is similar to the LSTM_Classifier. After finishing the training process, we get a cross-entropy plot as follows.

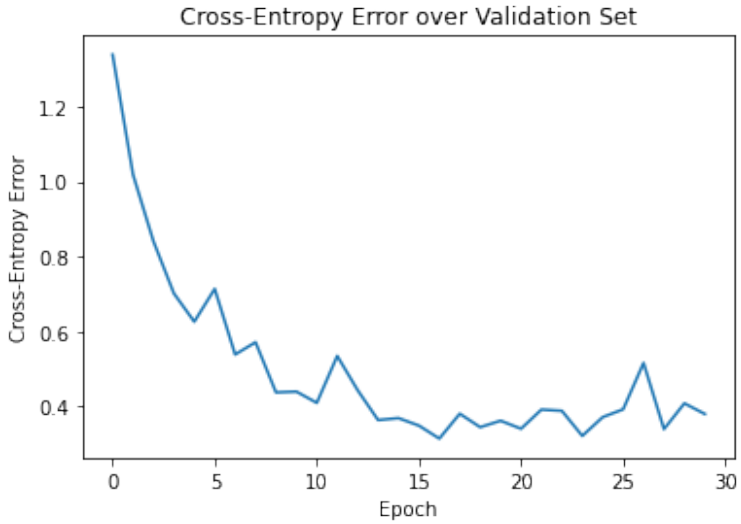


Figure 19. Cross entropy of validation dataset for GRU.

From figure 19, we can see that our GRU's performance on the validation dataset is significantly better than the LSTM. Furthermore, test accuracy is 89.0% which is significantly higher than both RNN and LSTM. Observing the loss plot it can be observed that it's again very stable compared to the recurrent layer and the network does learn correct predictions, hence it's safe to say that GRU also reduces the problem of vanishing gradients. Also, from figure 20 and 21, we can say that network can correctly learn the output patterns.

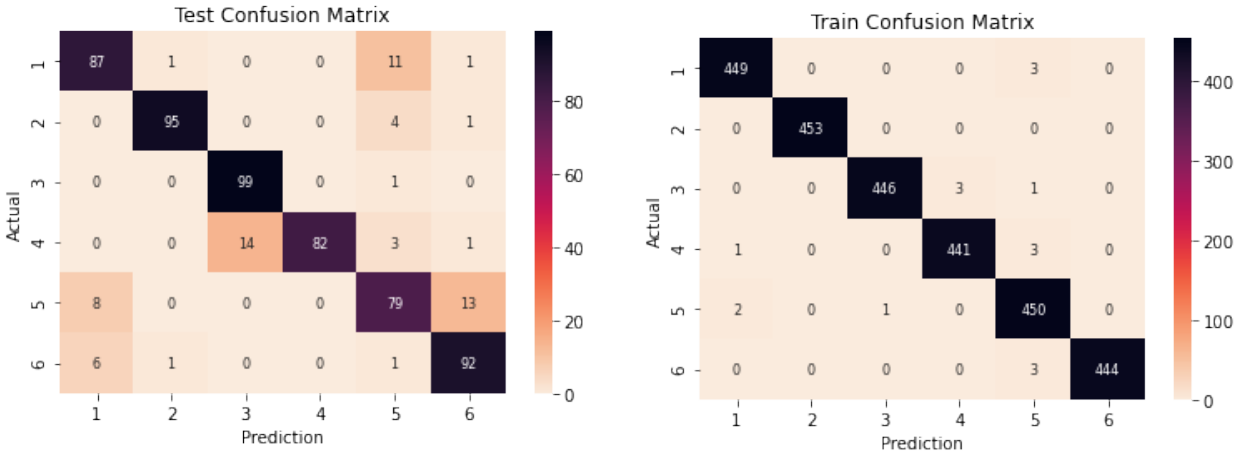


Figure 20, 21. Confusion matrix of test dataset and train dataset.

Overall, it can be said that for the human activity classification task, GRU seems to be the preferable network as it provides higher efficiency along with higher accuracy.

Output of Question 2

Convolutional Networks

So far we have worked with deep fully-connected networks, using them to explore different optimization strategies and network architectures. Fully-connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

In [8]:
```
%load_ext cython
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerical_gradient
from cs231n.layers import *
from cs231n.fast_layers import *
from cs231n.solver import Solver

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

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

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

```
The cython extension is already loaded. To reload it, use:
  %reload_ext cython
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

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,)
```

Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file `cs231n/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

You can test your implementation by running the following:

In [3]:
```
x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)

conv_param = {'stride': 2, 'pad': 1}
out, _ = conv_forward_naive(x, w, b, conv_param)
correct_out = np.array([[[[-0.08759809, -0.10987781],
                          [-0.18387192, -0.2109216 ]],
                         [[ 0.21027089,  0.21661097],
                          [ 0.22847626,  0.23004637]],
                         [[ 0.50813986,  0.54309974],
                          [ 0.64082444,  0.67101435]]],
                        [[[-0.98053589, -1.03143541],
                          [-1.19128892, -1.24695841]],
                         [[ 0.69108355,  0.66880383],
                          [ 0.59480972,  0.56776003]],
                         [[ 2.36270298,  2.36904306],
                          [ 2.38090835,  2.38247847]]]])

# Compare your output to ours; difference should be around e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

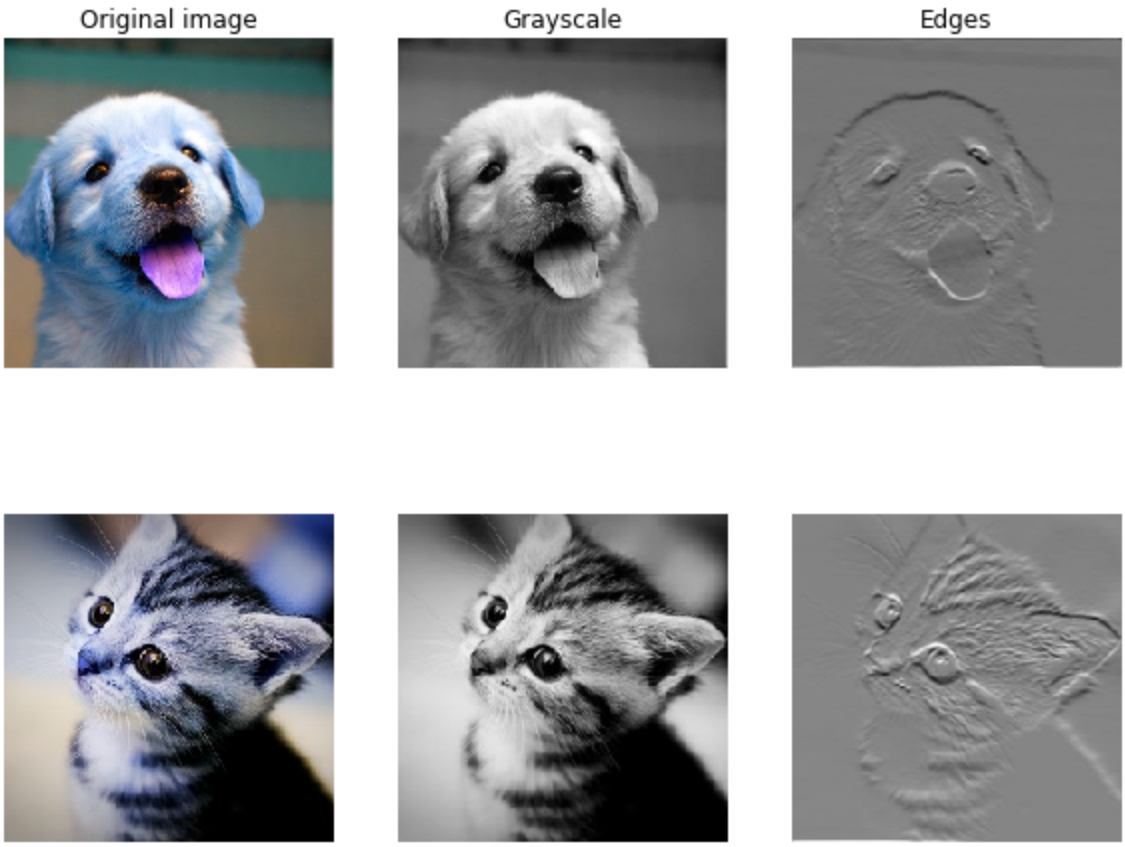
```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

Aside: Image processing via convolutions

As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

In [4]:
```
from cv2 import imread, resize

kitten, puppy = imread('kitten.jpg'), imread('puppy.jpg')
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = resize(puppy, (img_size, img_size)).transpose((2, 0, 1))
x[1, :, :, :] = resize(kitten_cropped, (img_size, img_size)).transpose((2, 0, 1))

# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))
```

```
# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = np.array([0, 128])

# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

def imshow_noax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_noax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_noax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_noax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_noax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_noax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```



Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cs231n/layers.py`. Again, you don't need to worry too much about computational efficiency.

When you are done, run the following to check your backward pass with a numeric gradient check.

In [5]:

```
np.random.seed(231)
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b, conv_param)[(
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_param)[(
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_param)[(

out, cache = conv_forward_naive(x, w, b, conv_param)
dx, dw, db = conv_backward_naive(dout, cache)

# Your errors should be around e-8 or less.
print('Testing conv_backward_naive function')
print('dx error: ', rel_error(dx, dx_num))
print('dw error: ', rel_error(dw, dw_num))
print('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  1.159803161159293e-08
dw error:  2.2471264748452487e-10
db error:  3.37264006649648e-11
```

Max-Pooling: Naive forward

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs231n/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

In [6]:
```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.26315789, -0.24842105],
                          [-0.20421053, -0.18947368]],
                         [[-0.14526316, -0.13052632],
                          [-0.08631579, -0.07157895]],
                         [[-0.02736842, -0.01263158],
                          [ 0.03157895,  0.04631579]]],
                        [[[ 0.09052632,  0.10526316],
                          [ 0.14947368,  0.16421053]],
                         [[ 0.20842105,  0.22315789],
                          [ 0.26736842,  0.28210526]],
                         [[ 0.32631579,  0.34105263],
                          [ 0.38526316,  0.4       ]]]])

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

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

Max-Pooling: Naive backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cs231n/layers.py`. You don't need to worry about computational efficiency.

Check your implementation with numeric gradient checking by running the following:

In [7]:
```
np.random.seed(231)
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x, pool_param)[0],

out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)

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

```
Testing max_pool_backward_naive function:
dx error:  3.27562514223145e-12
```

Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs231n/fast_layers.py` .

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs231n` directory:

```
python setup.py build_ext --inplace
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the following:

In [8]:

```
# Rel errors should be around e-9 or less
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(231)
x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

t0 = time()
out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
t1 = time()
out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
t2 = time()

print('Testing conv_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
t1 = time()
dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting conv_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
print('dw difference: ', rel_error(dw_naive, dw_fast))
print('db difference: ', rel_error(db_naive, db_fast))
```

```
Testing conv_forward_fast:
Naive: 4.745737s
Fast: 0.016643s
Speedup: 285.148325x
Difference:  4.926407851494105e-11
```

```
Testing conv_backward_fast:
Naive: 8.457416s
Fast: 0.011362s
Speedup: 744.370454x
dx difference:  1.949764775345631e-11
dw difference:  4.659623564096585e-13
db difference:  0.0
```

In [9]:

```
# Relative errors should be close to 0.0
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(231)
x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time()
out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
t1 = time()
out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
t2 = time()

print('Testing pool_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting pool_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.168703s
fast: 0.002104s
speedup: 80.180397x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.376410s
fast: 0.012205s
speedup: 30.840307x
dx difference:  0.0
```

Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `cs231n/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks.

In [10]:

```
from cs231n.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
np.random.seed(231)
x = np.random.randn(2, 3, 16, 16)
```

```
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
dx, dw, db = conv_relu_pool_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w, b, conv_para
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, conv_para
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, conv_para

# Relative errors should be around e-8 or less
print('Testing conv_relu_pool')
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 conv_relu_pool
dx error:  5.828178746516271e-09
dw error:  8.443628091870788e-09
db error:  3.57960501324485e-10
```

In [11]:

```
from cs231n.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_relu_forward(x, w, b, conv_param)
dx, dw, db = conv_relu_backward(dout, cache)

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

# Relative errors should be around e-8 or less
print('Testing conv_relu:')
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 conv_relu:
dx error:  3.5600610115232832e-09
dw error:  2.2497700915729298e-10
db error:  1.3087619975802167e-10
```

Three-layer ConvNet

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network.

Open the file `cs231n/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

Sanity check loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization this should go up.

In [12]:
```
model = ThreeLayerConvNet()

N = 50
X = np.random.randn(N, 3, 32, 32)
y = np.random.randint(10, size=N)

loss, grads = model.loss(X, y)
print('Initial loss (no regularization): ', loss)

model.reg = 0.5
loss, grads = model.loss(X, y)
print('Initial loss (with regularization): ', loss)
```

```
Initial loss (no regularization):  2.302586071243987
Initial loss (with regularization):  2.508255638232932
```

Gradient check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer. Note: correct implementations may still have relative errors up to the order of e-2.

In [13]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
np.random.seed(231)
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
                          dtype=np.float64)
loss, grads = model.loss(X, y)
# Errors should be small, but correct implementations may have
# relative errors up to the order of e-2
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verbose=False, h
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[parar
```

```
W1 max relative error: 1.380104e-04
W2 max relative error: 1.822723e-02
W3 max relative error: 3.064049e-04
b1 max relative error: 3.477652e-05
b2 max relative error: 2.516375e-03
b3 max relative error: 7.945660e-10
```

Overfit small data

A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

In [14]:

```
np.random.seed(231)

num_train = 100
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'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
                num_epochs=15, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

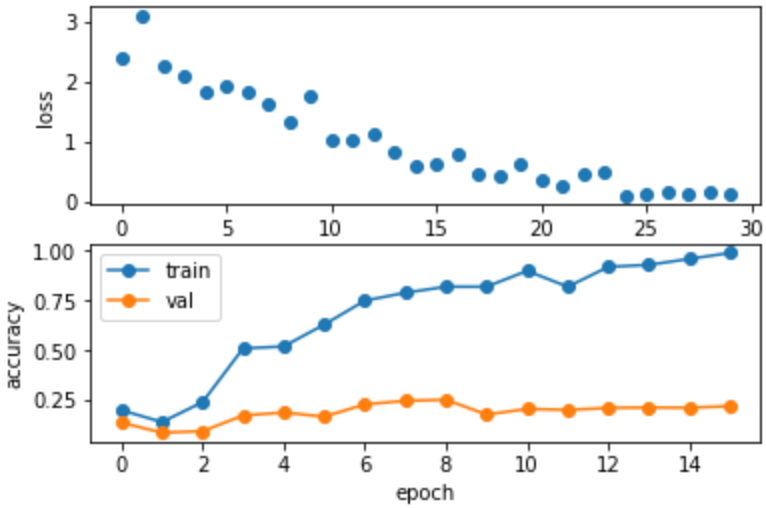
```
(Iteration 1 / 30) loss: 2.414060
(Epoch 0 / 15) train acc: 0.200000; val_acc: 0.137000
(Iteration 2 / 30) loss: 3.102925
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.270330
(Iteration 4 / 30) loss: 2.096705
(Epoch 2 / 15) train acc: 0.240000; val_acc: 0.094000
(Iteration 5 / 30) loss: 1.838880
(Iteration 6 / 30) loss: 1.934188
(Epoch 3 / 15) train acc: 0.510000; val_acc: 0.173000
(Iteration 7 / 30) loss: 1.827912
(Iteration 8 / 30) loss: 1.639574
(Epoch 4 / 15) train acc: 0.520000; val_acc: 0.188000
(Iteration 9 / 30) loss: 1.330082
(Iteration 10 / 30) loss: 1.756115
(Epoch 5 / 15) train acc: 0.630000; val_acc: 0.167000
(Iteration 11 / 30) loss: 1.024162
(Iteration 12 / 30) loss: 1.041826
(Epoch 6 / 15) train acc: 0.750000; val_acc: 0.229000
(Iteration 13 / 30) loss: 1.142777
(Iteration 14 / 30) loss: 0.835706
(Epoch 7 / 15) train acc: 0.790000; val_acc: 0.247000
(Iteration 15 / 30) loss: 0.587786
(Iteration 16 / 30) loss: 0.645509
(Epoch 8 / 15) train acc: 0.820000; val_acc: 0.252000
(Iteration 17 / 30) loss: 0.786844
(Iteration 18 / 30) loss: 0.467054
(Epoch 9 / 15) train acc: 0.820000; val_acc: 0.178000
(Iteration 19 / 30) loss: 0.429880
(Iteration 20 / 30) loss: 0.635498
(Epoch 10 / 15) train acc: 0.900000; val_acc: 0.206000
(Iteration 21 / 30) loss: 0.365807
(Iteration 22 / 30) loss: 0.284220
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.201000
(Iteration 23 / 30) loss: 0.469343
(Iteration 24 / 30) loss: 0.509369
(Epoch 12 / 15) train acc: 0.920000; val_acc: 0.211000
(Iteration 25 / 30) loss: 0.111638
(Iteration 26 / 30) loss: 0.145388
(Epoch 13 / 15) train acc: 0.930000; val_acc: 0.213000
(Iteration 27 / 30) loss: 0.155575
(Iteration 28 / 30) loss: 0.143398
(Epoch 14 / 15) train acc: 0.960000; val_acc: 0.212000
(Iteration 29 / 30) loss: 0.158160
```

```
(Iteration 30 / 30) loss: 0.118934
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.220000
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

In [15]:

```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



Train the net

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

In [16]:

```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

```
(Iteration 1 / 980) loss: 2.304740
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.098229
(Iteration 41 / 980) loss: 1.949788
(Iteration 61 / 980) loss: 1.888398
(Iteration 81 / 980) loss: 1.877093
(Iteration 101 / 980) loss: 1.851877
(Iteration 121 / 980) loss: 1.859353
(Iteration 141 / 980) loss: 1.800181
(Iteration 161 / 980) loss: 2.143292
(Iteration 181 / 980) loss: 1.830573
```

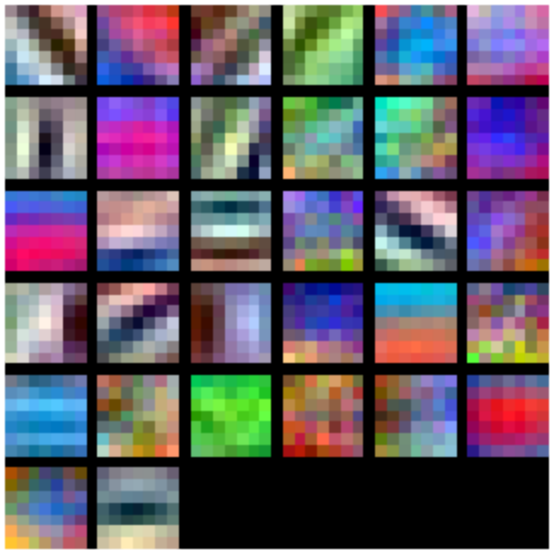
```
(Iteration 201 / 980) loss: 2.037280
(Iteration 221 / 980) loss: 2.020304
(Iteration 241 / 980) loss: 1.823728
(Iteration 261 / 980) loss: 1.692679
(Iteration 281 / 980) loss: 1.882594
(Iteration 301 / 980) loss: 1.798261
(Iteration 321 / 980) loss: 1.851960
(Iteration 341 / 980) loss: 1.716323
(Iteration 361 / 980) loss: 1.897655
(Iteration 381 / 980) loss: 1.319744
(Iteration 401 / 980) loss: 1.738790
(Iteration 421 / 980) loss: 1.488866
(Iteration 441 / 980) loss: 1.718409
(Iteration 461 / 980) loss: 1.744440
(Iteration 481 / 980) loss: 1.605460
(Iteration 501 / 980) loss: 1.494847
(Iteration 521 / 980) loss: 1.835179
(Iteration 541 / 980) loss: 1.483923
(Iteration 561 / 980) loss: 1.676871
(Iteration 581 / 980) loss: 1.438325
(Iteration 601 / 980) loss: 1.443469
(Iteration 621 / 980) loss: 1.529369
(Iteration 641 / 980) loss: 1.763475
(Iteration 661 / 980) loss: 1.790329
(Iteration 681 / 980) loss: 1.693343
(Iteration 701 / 980) loss: 1.637078
(Iteration 721 / 980) loss: 1.644564
(Iteration 741 / 980) loss: 1.708919
(Iteration 761 / 980) loss: 1.494252
(Iteration 781 / 980) loss: 1.901751
(Iteration 801 / 980) loss: 1.898991
(Iteration 821 / 980) loss: 1.489988
(Iteration 841 / 980) loss: 1.377615
(Iteration 861 / 980) loss: 1.763751
(Iteration 881 / 980) loss: 1.540284
(Iteration 901 / 980) loss: 1.525582
(Iteration 921 / 980) loss: 1.674166
(Iteration 941 / 980) loss: 1.714316
(Iteration 961 / 980) loss: 1.534668
(Epoch 1 / 1) train acc: 0.504000; val_acc: 0.499000
```

Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

In [17]:

```
from cs231n.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```



Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully-connected networks. As proposed in the original paper [3], batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

Normally batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization needs to accept inputs of shape `(N, C, H, W)` and produce outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

If the feature map was produced using convolutions, then we expect the statistics of each feature channel to be relatively consistent both between different images and different locations within the same image. Therefore spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over both the minibatch dimension `N` and the spatial dimensions `H` and `W`.

[3] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.

Spatial batch normalization: forward

In the file `cs231n/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

In [18]:

```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 3, 4, 5
x = 4 * np.random.randn(N, C, H, W) + 10

print('Before spatial batch normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x.mean(axis=(0, 2, 3)))
print('  Stds: ', x.std(axis=(0, 2, 3)))
```

```
# Means should be close to zero and stds close to one
gamma, beta = np.ones(C), np.zeros(C)
bn_param = {'mode': 'train'}
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
gamma, beta = np.asarray([3, 4, 5]), np.asarray([6, 7, 8])
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization (nontrivial gamma, beta):')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))
```

```
Before spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [9.33463814 8.90909116 9.11056338]
  Stds:  [3.61447857 3.19347686 3.5168142 ]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 5.85642645e-16  5.93969318e-16 -8.88178420e-17]
  Stds:  [0.99999962 0.99999951 0.9999996 ]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999885 3.99999804 4.99999798]
```

In [19]:

```
np.random.seed(231)
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After spatial batch normalization (test-time):')
print('  means: ', a_norm.mean(axis=(0, 2, 3)))
print('  stds: ', a_norm.std(axis=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  [-0.08034406  0.07562881  0.05716371  0.04378383]
  stds:  [0.96718744 1.0299714  1.02887624 1.00585577]
```

Spatial batch normalization: backward

In the file `cs231n/layers.py` , implement the backward pass for spatial batch normalization in the function `spatial_batchnorm_backward` . Run the following to check your implementation using a numeric gradient check:

In [20]:

```
np.random.seed(231)
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

bn_param = {'mode': 'train'}
fx = lambda x: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda b: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

#You should expect errors of magnitudes between 1e-12~1e-06
_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

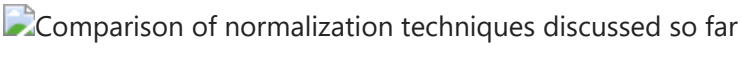
```
dx error:  3.083846820796372e-07
dgamma error:  7.09738489671469e-12
dbeta error:  3.275608725278405e-12
```

Group Normalization

In the previous notebook, we mentioned that Layer Normalization is an alternative normalization technique that mitigates the batch size limitations of Batch Normalization. However, as the authors of [4] observed, Layer Normalization does not perform as well as Batch Normalization when used with Convolutional Layers:

> With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction, and re-centering and rescaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

The authors of [5] propose an intermediary technique. In contrast to Layer Normalization, where you normalize over the entire feature per-datapoint, they suggest a consistent splitting of each per-datapoint feature into G groups, and a per-group per-datapoint normalization instead.


Comparison of normalization techniques discussed so far

Visual comparison of the normalization techniques discussed so far (image edited from [5])

Even though an assumption of equal contribution is still being made within each group, the authors hypothesize that this is not as problematic, as innate grouping arises within features for visual recognition. One example they use to illustrate this is that many high-performance handcrafted features in traditional Computer Vision have terms that are explicitly grouped together. Take for example Histogram of Oriented Gradients [6]-- after computing histograms per spatially local block, each per-block histogram is normalized before being concatenated together to form the final feature vector.

You will now implement Group Normalization. Note that this normalization technique that you are to implement in the following cells was introduced and published to arXiv *less than a month ago* -- this truly is still an ongoing and excitingly active field of research!

[4] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.

[5] Wu, Yuxin, and Kaiming He. "Group Normalization." arXiv preprint arXiv:1803.08494 (2018).

[6] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.

Group normalization: forward

In the file `cs231n/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

In [21]:
```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 6, 4, 5
G = 2
x = 4 * np.random.randn(N, C, H, W) + 10
x_g = x.reshape((N*G,-1))
print('Before spatial group normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x_g.mean(axis=1))
print('  Stds: ', x_g.std(axis=1))

# Means should be close to zero and stds close to one
gamma, beta = np.ones((1,C,1,1)), np.zeros((1,C,1,1))
bn_param = {'mode': 'train'}

out, _ = spatial_groupnorm_forward(x, gamma, beta, G, bn_param)
out_g = out.reshape((N*G,-1))
print('After spatial group normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out_g.mean(axis=1))
print('  Stds: ', out_g.std(axis=1))
```

```
Before spatial group normalization:
  Shape:  (2, 6, 4, 5)
  Means:  [9.72505327 8.51114185 8.9147544  9.43448077]
  Stds:  [3.67070958 3.09892597 4.27043622 3.97521327]
After spatial group normalization:
  Shape:  (1, 1, 1, 2, 6, 4, 5)
  Means:  [-2.14643118e-16  5.25505565e-16  2.58126853e-16 -3.62672855e-16]
  Stds:  [0.99999963 0.99999948 0.99999973 0.99999968]
```

Spatial group normalization: backward

In the file `cs231n/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_groupnorm_backward`. Run the following to check your implementation using a numeric gradient check:

In [22]:
```
np.random.seed(231)
N, C, H, W = 2, 6, 4, 5
G = 2
```

```
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

gn_param = {}
fx = lambda x: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fg = lambda a: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fb = lambda b: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

_, cache = spatial_groupnorm_forward(x, gamma, beta, G, gn_param)
dx, dgamma, dbeta = spatial_groupnorm_backward(dout, cache)
#You should expect errors of magnitudes between 1e-12~1e-07
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  6.34590431845254e-08
dgamma error:  1.0546047434202244e-11
dbeta error:  3.810857316122484e-12
```

In [ ]:

What's this PyTorch business?

You've written a lot of code in this assignment to provide a whole host of neural network functionality. Dropout, Batch Norm, and 2D convolutions are some of the workhorses of deep learning in computer vision. You've also worked hard to make your code efficient and vectorized.

For the last part of this assignment, though, we're going to leave behind your beautiful codebase and instead migrate to one of two popular deep learning frameworks: in this instance, PyTorch (or TensorFlow, if you switch over to that notebook).

What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

Why?

- Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

PyTorch versions

This notebook assumes that you are using **PyTorch version 0.4**. Prior to this version, Tensors had to be wrapped in Variable objects to be used in autograd; however Variables have now been deprecated. In addition 0.4 also separates a Tensor's datatype from its device, and uses numpy-style factories for constructing Tensors rather than directly invoking Tensor constructors.

How will I learn PyTorch?

Justin Johnson has made an excellent tutorial for PyTorch.

You can also find the detailed API doc here. If you have other questions that are not addressed by the API docs, the PyTorch forum is a much better place to ask than StackOverflow.

Table of Contents

This assignment has 5 parts. You will learn PyTorch on different levels of abstractions, which will help you understand it better and prepare you for the final project.

1. Preparation: we will use CIFAR-10 dataset.

2. Barebones PyTorch: we will work directly with the lowest-level PyTorch Tensors.
3. PyTorch Module API: we will use `nn.Module` to define arbitrary neural network architecture.
4. PyTorch Sequential API: we will use `nn.Sequential` to define a linear feed-forward network very conveniently.
5. CIFAR-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CIFAR-10. You can experiment with any layer, optimizer, hyperparameters or other advanced features.

Here is a table of comparison:

API	Flexibility	Convenience
Barebone	High	Low
`nn.Module`	High	Medium
`nn.Sequential`	Low	High

Part I. Preparation

First, we load the CIFAR-10 dataset. This might take a couple minutes the first time you do it, but the files should stay cached after that.

In previous parts of the assignment we had to write our own code to download the CIFAR-10 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

import torchvision.datasets as dset
import torchvision.transforms as T

import numpy as np
```

```
C:\Users\ataka\anaconda3\lib\site-packages\torchvision\io\image.py:11: UserWarning: Failed
to load image Python extension: Could not find module 'C:\Users\ataka\anaconda3\Lib\site-p
ackages\torchvision\image.pyd' (or one of its dependencies). Try using the full path with
constructor syntax.
  warn(f"Failed to load image Python extension: {e}")
```

In [2]:
```
NUM_TRAIN = 49000

# The torchvision.transforms package provides tools for preprocessing data
# and for performing data augmentation; here we set up a transform to
# preprocess the data by subtracting the mean RGB value and dividing by the
# standard deviation of each RGB value; we've hardcoded the mean and std.
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
            ])

# We set up a Dataset object for each split (train / val / test); Datasets load
# training examples one at a time, so we wrap each Dataset in a DataLoader which
# iterates through the Dataset and forms minibatches. We divide the CIFAR-10
# training set into train and val sets by passing a Sampler object to the
```

```
# DataLoader telling how it should sample from the underlying Dataset.
cifar10_train = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                             transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

cifar10_val = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN, 50000)))

cifar10_test = dset.CIFAR10('./cs231n/datasets', train=False, download=True,
                            transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

You have an option to **use GPU by setting the flag to True below**. It is not necessary to use GPU for this assignment. Note that if your computer does not have CUDA enabled, `torch.cuda.is_available()` will return False and this notebook will fallback to CPU mode.

The global variables `dtype` and `device` will control the data types throughout this assignment.

In [3]:

```
USE_GPU = True

dtype = torch.float32 # we will be using float throughout this tutorial

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss
print_every = 100

print('using device:', device)
```

```
using device: cuda
```

Part II. Barebones PyTorch

PyTorch ships with high-level APIs to help us define model architectures conveniently, which we will cover in Part II of this tutorial. In this section, we will start with the barebone PyTorch elements to understand the autograd engine better. After this exercise, you will come to appreciate the high-level model API more.

We will start with a simple fully-connected ReLU network with two hidden layers and no biases for CIFAR classification. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. It is important that you understand every line, because you will write a harder version after the example.

When we create a PyTorch Tensor with `requires_grad=True`, then operations involving that Tensor will not just compute values; they will also build up a computational graph in the background, allowing us to easily backpropagate through the graph to compute gradients of some Tensors with respect to a downstream loss. Concretely if x is a Tensor with `x.requires_grad == True` then after backpropagation `x.grad` will be another Tensor holding the gradient of x with respect to the scalar loss at the end.

PyTorch Tensors: Flatten Function

A PyTorch Tensor is conceptionally similar to a numpy array: it is an n-dimensional grid of numbers, and like numpy PyTorch provides many functions to efficiently operate on Tensors. As a simple example, we provide a `flatten` function below which reshapes image data for use in a fully-connected neural network.

Recall that image data is typically stored in a Tensor of shape N x C x H x W, where:

- N is the number of datapoints
- C is the number of channels
- H is the height of the intermediate feature map in pixels
- W is the height of the intermediate feature map in pixels

This is the right way to represent the data when we are doing something like a 2D convolution, that needs spatial understanding of where the intermediate features are relative to each other. When we use fully connected affine layers to process the image, however, we want each datapoint to be represented by a single vector -- it's no longer useful to segregate the different channels, rows, and columns of the data. So, we use a "flatten" operation to collapse the `C x H x W` values per representation into a single long vector. The flatten function below first reads in the N, C, H, and W values from a given batch of data, and then returns a "view" of that data. "View" is analogous to numpy's "reshape" method: it reshapes x's dimensions to be N x ??, where ?? is allowed to be anything (in this case, it will be C x H x W, but we don't need to specify that explicitly).

In [4]:
```
def flatten(x):
    N = x.shape[0] # read in N, C, H, W
    return x.view(N, -1)  # "flatten" the C * H * W values into a single vector per image

def test_flatten():
    x = torch.arange(12).view(2, 1, 3, 2)
    print('Before flattening: ', x)
    print('After flattening: ', flatten(x))

test_flatten()
```

```
Before flattening:  tensor([[[[ 0,  1],
          [ 2,  3],
          [ 4,  5]]],


        [[[ 6,  7],
          [ 8,  9],
          [10, 11]]]])
After flattening:  tensor([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11]])
```

Barebones PyTorch: Two-Layer Network

Here we define a function `two_layer_fc` which performs the forward pass of a two-layer fully-connected ReLU network on a batch of image data. After defining the forward pass we check that it doesn't crash and that it produces outputs of the right shape by running zeros through the network.

You don't have to write any code here, but it's important that you read and understand the implementation.

In [5]:
```
import torch.nn.functional as F  # useful stateless functions

def two_layer_fc(x, params):
    """
```

```
    A fully-connected neural networks; the architecture is:
    NN is fully connected -> ReLU -> fully connected layer.
    Note that this function only defines the forward pass;
    PyTorch will take care of the backward pass for us.

    The input to the network will be a minibatch of data, of shape
    (N, d1, ..., dM) where d1 * ... * dM = D. The hidden layer will have H units,
    and the output layer will produce scores for C classes.

    Inputs:
    - x: A PyTorch Tensor of shape (N, d1, ..., dM) giving a minibatch of
      input data.
    - params: A list [w1, w2] of PyTorch Tensors giving weights for the network;
      w1 has shape (D, H) and w2 has shape (H, C).

    Returns:
    - scores: A PyTorch Tensor of shape (N, C) giving classification scores for
      the input data x.
    """
    # first we flatten the image
    x = flatten(x)  # shape: [batch_size, C x H x W]

    w1, w2 = params

    # Forward pass: compute predicted y using operations on Tensors. Since w1 and
    # w2 have requires_grad=True, operations involving these Tensors will cause
    # PyTorch to build a computational graph, allowing automatic computation of
    # gradients. Since we are no longer implementing the backward pass by hand we
    # don't need to keep references to intermediate values.
    # you can also use `.clamp(min=0)`, equivalent to F.relu()
    x = F.relu(x.mm(w1))
    x = x.mm(w2)
    return x


def two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature dimension 50
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 10]

two_layer_fc_test()
```

```
torch.Size([64, 10])
```

Barebones PyTorch: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet`, which will perform the forward pass of a three-layer convolutional network. Like above, we can immediately test our implementation by passing zeros through the network. The network should have the following architecture:

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for C classes.

HINT: For convolutions: http://pytorch.org/docs/stable/nn.html#torch.nn.functional.conv2d; pay attention to the shapes of convolutional filters!

In [6]:

```
def three_layer_convnet(x, params):
    """
    Performs the forward pass of a three-layer convolutional network with the
    architecture defined above.

    Inputs:
    - x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch of images
    - params: A list of PyTorch Tensors giving the weights and biases for the
      network; should contain the following:
      - conv_w1: PyTorch Tensor of shape (channel_1, 3, KH1, KW1) giving weights
        for the first convolutional layer
      - conv_b1: PyTorch Tensor of shape (channel_1,) giving biases for the first
        convolutional layer
      - conv_w2: PyTorch Tensor of shape (channel_2, channel_1, KH2, KW2) giving
        weights for the second convolutional layer
      - conv_b2: PyTorch Tensor of shape (channel_2,) giving biases for the second
        convolutional layer
      - fc_w: PyTorch Tensor giving weights for the fully-connected layer. Can you
        figure out what the shape should be?
      - fc_b: PyTorch Tensor giving biases for the fully-connected layer. Can you
        figure out what the shape should be?

    Returns:
    - scores: PyTorch Tensor of shape (N, C) giving classification scores for x
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None
    ################################################################################
    # TODO: Implement the forward pass for the three-layer ConvNet.                #
    ################################################################################

    conv1 = F.conv2d(x, weight=conv_w1, bias=conv_b1, padding=2)
    relu1 = F.relu(conv1)
    conv2 = F.conv2d(relu1, weight=conv_w2, bias=conv_b2, padding=1)
    relu2 = F.relu(conv2)
    relu2_flat = flatten(relu2)
    scores = relu2_flat.mm(fc_w) + fc_b

    ################################################################################
    #                                 END OF YOUR CODE                             #
    ################################################################################
    return scores
```

After defining the forward pass of the ConvNet above, run the following cell to test your implementation.

When you run this function, scores should have shape (64, 10).

In [7]:

```
def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image size [3, 32,

    conv_w1 = torch.zeros((6, 3, 5, 5), dtype=dtype)  # [out_channel, in_channel, kernel_H
    conv_b1 = torch.zeros((6,))  # out_channel
    conv_w2 = torch.zeros((9, 6, 3, 3), dtype=dtype)  # [out_channel, in_channel, kernel_H
    conv_b2 = torch.zeros((9,))  # out_channel

    # you must calculate the shape of the tensor after two conv layers, before the fully-c
    fc_w = torch.zeros((9 * 32 * 32, 10))
    fc_b = torch.zeros(10)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b])
```

```
    print(scores.size())  # you should see [64, 10]
three_layer_convnet_test()
```

```
torch.Size([64, 10])
```

Barebones PyTorch: Initialization

Let's write a couple utility methods to initialize the weight matrices for our models.

- `random_weight(shape)` initializes a weight tensor with the Kaiming normalization method.
- `zero_weight(shape)` initializes a weight tensor with all zeros. Useful for instantiating bias parameters.

The `random_weight` function uses the Kaiming normal initialization method, described in:

He et al, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, ICCV 2015, https://arxiv.org/abs/1502.01852

In [8]:
```
def random_weight(shape):
    """
    Create random Tensors for weights; setting requires_grad=True means that we
    want to compute gradients for these Tensors during the backward pass.
    We use Kaiming normalization: sqrt(2 / fan_in)
    """
    if len(shape) == 2:  # FC weight
        fan_in = shape[0]
    else:
        fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_channel, kH, kW]
    # randn is standard normal distribution generator.
    w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fan_in)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, device=device, dtype=dtype, requires_grad=True)

# create a weight of shape [3 x 5]
# you should see the type `torch.cuda.FloatTensor` if you use GPU.
# Otherwise it should be `torch.FloatTensor`
random_weight((3, 5))
```

Out[8]:
```
tensor([[ 0.2943,  0.1578, -0.2573,  0.6830, -0.3399],
        [ 0.2941,  0.2806,  0.0935, -0.4633, -0.0477],
        [-0.2860, -0.7216,  0.9535, -1.0970, -0.4771]], device='cuda:0',
       requires_grad=True)
```

Barebones PyTorch: Check Accuracy

When training the model we will use the following function to check the accuracy of our model on the training or validation sets.

When checking accuracy we don't need to compute any gradients; as a result we don't need PyTorch to build a computational graph for us when we compute scores. To prevent a graph from being built we scope our computation under a `torch.no_grad()` context manager.

In [9]:
```
def check_accuracy_part2(loader, model_fn, params):
    """
    Check the accuracy of a classification model.

    Inputs:
    - loader: A DataLoader for the data split we want to check
```

```
    - model_fn: A function that performs the forward pass of the model,
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the model

    Returns: Nothing, but prints the accuracy of the model
    """
    split = 'val' if loader.dataset.train else 'test'
    print('Checking accuracy on the %s set' % split)
    num_correct, num_samples = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.int64)
            scores = model_fn(x, params)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 * acc))
```

BareBones PyTorch: Training Loop

We can now set up a basic training loop to train our network. We will train the model using stochastic gradient descent without momentum. We will use `torch.functional.cross_entropy` to compute the loss; you can read about it here.

The training loop takes as input the neural network function, a list of initialized parameters (`[w1, w2]` in our example), and learning rate.

In [10]:

```
def train_part2(model_fn, params, learning_rate):
    """
    Train a model on CIFAR-10.

    Inputs:
    - model_fn: A Python function that performs the forward pass of the model.
      It should have the signature scores = model_fn(x, params) where x is a
      PyTorch Tensor of image data, params is a list of PyTorch Tensors giving
      model weights, and scores is a PyTorch Tensor of shape (N, C) giving
      scores for the elements in x.
    - params: List of PyTorch Tensors giving weights for the model
    - learning_rate: Python scalar giving the learning rate to use for SGD

    Returns: Nothing
    """
    for t, (x, y) in enumerate(loader_train):
        # Move the data to the proper device (GPU or CPU)
        x = x.to(device=device, dtype=dtype)
        y = y.to(device=device, dtype=torch.long)

        # Forward pass: compute scores and loss
        scores = model_fn(x, params)
        loss = F.cross_entropy(scores, y)

        # Backward pass: PyTorch figures out which Tensors in the computational
        # graph has requires_grad=True and uses backpropagation to compute the
        # gradient of the loss with respect to these Tensors, and stores the
        # gradients in the .grad attribute of each Tensor.
        loss.backward()

        # Update parameters. We don't want to backpropagate through the
        # parameter updates, so we scope the updates under a torch.no_grad()
        # context manager to prevent a computational graph from being built.
```

```
        with torch.no_grad():
            for w in params:
                w -= learning_rate * w.grad

                # Manually zero the gradients after running the backward pass
                w.grad.zero_()

        if t % print_every == 0:
            print('Iteration %d, loss = %.4f' % (t, loss.item()))
            check_accuracy_part2(loader_val, model_fn, params)
            print()
```

BareBones PyTorch: Train a Two-Layer Network

Now we are ready to run the training loop. We need to explicitly allocate tensors for the fully connected weights, `w1` and `w2`.

Each minibatch of CIFAR has 64 examples, so the tensor shape is `[64, 3, 32, 32]`.

After flattening, `x` shape should be `[64, 3 * 32 * 32]`. This will be the size of the first dimension of `w1`. The second dimension of `w1` is the hidden layer size, which will also be the first dimension of `w2`.

Finally, the output of the network is a 10-dimensional vector that represents the probability distribution over 10 classes.

You don't need to tune any hyperparameters but you should see accuracies above 40% after training for one epoch.

In [11]:
```
hidden_layer_size = 4000
learning_rate = 1e-2

w1 = random_weight((3 * 32 * 32, hidden_layer_size))
w2 = random_weight((hidden_layer_size, 10))

train_part2(two_layer_fc, [w1, w2], learning_rate)
```

```
Iteration 0, loss = 3.4706
Checking accuracy on the val set
Got 182 / 1000 correct (18.20%)

Iteration 100, loss = 2.3327
Checking accuracy on the val set
Got 390 / 1000 correct (39.00%)

Iteration 200, loss = 1.6982
Checking accuracy on the val set
Got 427 / 1000 correct (42.70%)

Iteration 300, loss = 2.1812
Checking accuracy on the val set
Got 413 / 1000 correct (41.30%)

Iteration 400, loss = 1.7606
Checking accuracy on the val set
Got 438 / 1000 correct (43.80%)

Iteration 500, loss = 1.9049
Checking accuracy on the val set
Got 399 / 1000 correct (39.90%)

Iteration 600, loss = 1.7482
```

```
Checking accuracy on the val set
Got 434 / 1000 correct (43.40%)

Iteration 700, loss = 1.8573
Checking accuracy on the val set
Got 456 / 1000 correct (45.60%)
```

BareBones PyTorch: Training a ConvNet

In the below you should use the functions defined above to train a three-layer convolutional network on CIFAR. The network should have the following architecture:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You don't need to tune any hyperparameters, but if everything works correctly you should achieve an accuracy above 42% after one epoch.

In [12]:

```
learning_rate = 3e-3

channel_1 = 32
channel_2 = 16

conv_w1 = None
conv_b1 = None
conv_w2 = None
conv_b2 = None
fc_w = None
fc_b = None

################################################################################
# TODO: Initialize the parameters of a three-layer ConvNet.                    #
################################################################################

conv_w1 = random_weight((channel_1, 3, 5, 5))
conv_b1 = zero_weight((channel_1,))
conv_w2 = random_weight((channel_2, 32, 3, 3))
conv_b2 = zero_weight((channel_2,))
fc_w = random_weight((channel_2*32*32, 10))
fc_b = zero_weight((10,))

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

params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
train_part2(three_layer_convnet, params, learning_rate)
```

```
Iteration 0, loss = 3.0929
Checking accuracy on the val set
Got 114 / 1000 correct (11.40%)

Iteration 100, loss = 1.9274
Checking accuracy on the val set
```

```
Got 365 / 1000 correct (36.50%)

Iteration 200, loss = 1.7626
Checking accuracy on the val set
Got 408 / 1000 correct (40.80%)

Iteration 300, loss = 1.9109
Checking accuracy on the val set
Got 442 / 1000 correct (44.20%)

Iteration 400, loss = 1.4008
Checking accuracy on the val set
Got 455 / 1000 correct (45.50%)

Iteration 500, loss = 1.6100
Checking accuracy on the val set
Got 452 / 1000 correct (45.20%)

Iteration 600, loss = 1.6901
Checking accuracy on the val set
Got 466 / 1000 correct (46.60%)

Iteration 700, loss = 1.6555
Checking accuracy on the val set
Got 485 / 1000 correct (48.50%)
```

Part III. PyTorch Module API

Barebone PyTorch requires that we track all the parameter tensors by hand. This is fine for small networks with a few tensors, but it would be extremely inconvenient and error-prone to track tens or hundreds of tensors in larger networks.

PyTorch provides the `nn.Module` API for you to define arbitrary network architectures, while tracking every learnable parameters for you. In Part II, we implemented SGD ourselves. PyTorch also provides the `torch.optim` package that implements all the common optimizers, such as RMSProp, Adagrad, and Adam. It even supports approximate second-order methods like L-BFGS! You can refer to the doc for the exact specifications of each optimizer.

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.

2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!

3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

After you define your Module subclass, you can instantiate it as an object and call it just like the NN forward function in part II.

Module API: Two-Layer Network

Here is a concrete example of a 2-layer fully connected network:

In [13]:
```
class TwoLayerFC(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        # assign layer objects to class attributes
        self.fc1 = nn.Linear(input_size, hidden_size)
        # nn.init package contains convenient initialization methods
        # http://pytorch.org/docs/master/nn.html#torch-nn-init
        nn.init.kaiming_normal_(self.fc1.weight)
        self.fc2 = nn.Linear(hidden_size, num_classes)
        nn.init.kaiming_normal_(self.fc2.weight)

    def forward(self, x):
        # forward always defines connectivity
        x = flatten(x)
        scores = self.fc2(F.relu(self.fc1(x)))
        return scores

def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64, feature dimensio
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_TwoLayerFC()
```

```
torch.Size([64, 10])
```

Module API: Three-Layer ConvNet

It's your turn to implement a 3-layer ConvNet followed by a fully connected layer. The network architecture should be the same as in Part II:

1. Convolutional layer with `channel_1` 5x5 filters with zero-padding of 2
2. ReLU
3. Convolutional layer with `channel_2` 3x3 filters with zero-padding of 1
4. ReLU
5. Fully-connected layer to `num_classes` classes

You should initialize the weight matrices of the model using the Kaiming normal initialization method.

HINT: http://pytorch.org/docs/stable/nn.html#conv2d

After you implement the three-layer ConvNet, the `test_ThreeLayerConvNet` function will run your implementation; it should print `(64, 10)` for the shape of the output scores.

In [14]:
```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        ########################################################################
        # TODO: Set up the layers you need for a three-layer ConvNet with the  #
        # architecture defined above.                                          #
        ########################################################################

        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2, bias=True)
        nn.init.kaiming_normal_(self.conv1.weight)
```

```
        nn.init.constant_(self.conv1.bias, 0)

        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1, bias=True)
        nn.init.kaiming_normal_(self.conv2.weight)
        nn.init.constant_(self.conv2.bias, 0)

        self.fc = nn.Linear(channel_2*32*32, num_classes)
        nn.init.kaiming_normal_(self.fc.weight)
        nn.init.constant_(self.fc.bias, 0)

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

    def forward(self, x):
        scores = None
        ############################################################################
        # TODO: Implement the forward function for a 3-layer ConvNet. you          #
        # should use the layers you defined in __init__ and specify the            #
        # connectivity of those layers in forward()                                #
        ############################################################################

        relu1 = F.relu(self.conv1(x))
        relu2 = F.relu(self.conv2(relu1))
        scores = self.fc(flatten(relu2))

        ############################################################################
        #                               END OF YOUR CODE                           #
        ############################################################################
        return scores


def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image size [3, 32,
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8, num_classes=10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_ThreeLayerConvNet()
```

```
torch.Size([64, 10])
```

Module API: Check Accuracy

Given the validation or test set, we can check the classification accuracy of a neural network.

This version is slightly different from the one in part II. You don't manually pass in the parameters anymore.

In [15]:

```
def check_accuracy_part34(loader, model):
    if loader.dataset.train:
        print('Checking accuracy on validation set')
    else:
        print('Checking accuracy on test set')
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
```

```
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 * acc))
```

Module API: Training Loop

We also use a slightly different training loop. Rather than updating the values of the weights ourselves, we use an Optimizer object from the `torch.optim` package, which abstract the notion of an optimization algorithm and provides implementations of most of the algorithms commonly used to optimize neural networks.

In [16]:

```
def train_part34(model, optimizer, epochs=1):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

    Inputs:
    - model: A PyTorch Module giving the model to train.
    - optimizer: An Optimizer object we will use to train the model
    - epochs: (Optional) A Python integer giving the number of epochs to train for

    Returns: Nothing, but prints model accuracies during training.
    """
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

            scores = model(x)
            loss = F.cross_entropy(scores, y)

            # Zero out all of the gradients for the variables which the optimizer
            # will update.
            optimizer.zero_grad()

            # This is the backwards pass: compute the gradient of the loss with
            # respect to each  parameter of the model.
            loss.backward()

            # Actually update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()

            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy_part34(loader_val, model)
                print()
```

Module API: Train a Two-Layer Network

Now we are ready to run the training loop. In contrast to part II, we don't explicitly allocate parameter tensors anymore.

Simply pass the input size, hidden layer size, and number of classes (i.e. output size) to the constructor of `TwoLayerFC`.

You also need to define an optimizer that tracks all the learnable parameters inside `TwoLayerFC`.

You don't need to tune any hyperparameters, but you should see model accuracies above 40% after training for one epoch.

In [17]:
```
hidden_layer_size = 4000
learning_rate = 1e-2
model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.9178
Checking accuracy on validation set
Got 169 / 1000 correct (16.90)

Iteration 100, loss = 2.3137
Checking accuracy on validation set
Got 336 / 1000 correct (33.60)

Iteration 200, loss = 1.7811
Checking accuracy on validation set
Got 379 / 1000 correct (37.90)

Iteration 300, loss = 1.7102
Checking accuracy on validation set
Got 350 / 1000 correct (35.00)

Iteration 400, loss = 2.0582
Checking accuracy on validation set
Got 390 / 1000 correct (39.00)

Iteration 500, loss = 1.9522
Checking accuracy on validation set
Got 389 / 1000 correct (38.90)

Iteration 600, loss = 1.9741
Checking accuracy on validation set
Got 387 / 1000 correct (38.70)

Iteration 700, loss = 1.8645
Checking accuracy on validation set
Got 428 / 1000 correct (42.80)
```

Module API: Train a Three-Layer ConvNet

You should now use the Module API to train a three-layer ConvNet on CIFAR. This should look very similar to training the two-layer network! You don't need to tune any hyperparameters, but you should achieve above above 45% after training for one epoch.

You should train the model using stochastic gradient descent without momentum.

In [18]:
```
learning_rate = 3e-3
channel_1 = 32
channel_2 = 16

model = None
optimizer = None
################################################################################
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
################################################################################

model = ThreeLayerConvNet(3, channel_1, channel_2, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

################################################################################
```

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.2008
Checking accuracy on validation set
Got 107 / 1000 correct (10.70)

Iteration 100, loss = 1.8061
Checking accuracy on validation set
Got 346 / 1000 correct (34.60)

Iteration 200, loss = 1.7921
Checking accuracy on validation set
Got 403 / 1000 correct (40.30)

Iteration 300, loss = 1.7425
Checking accuracy on validation set
Got 409 / 1000 correct (40.90)

Iteration 400, loss = 1.7028
Checking accuracy on validation set
Got 455 / 1000 correct (45.50)

Iteration 500, loss = 1.7719
Checking accuracy on validation set
Got 444 / 1000 correct (44.40)

Iteration 600, loss = 1.5905
Checking accuracy on validation set
Got 480 / 1000 correct (48.00)

Iteration 700, loss = 1.5565
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)
```

Part IV. PyTorch Sequential API

Part III introduced the PyTorch Module API, which allows you to define arbitrary learnable layers and their connectivity.

For simple models like a stack of feed forward layers, you still need to go through 3 steps: subclass `nn.Module`, assign layers to class attributes in `__init__`, and call each layer one by one in `forward()`. Is there a more convenient way?

Fortunately, PyTorch provides a container Module called `nn.Sequential`, which merges the above steps into one. It is not as flexible as `nn.Module`, because you cannot specify more complex topology than a feed-forward stack, but it's good enough for many use cases.

Sequential API: Two-Layer Network

Let's see how to rewrite our two-layer fully connected network example with `nn.Sequential`, and train it using the training loop defined above.

Again, you don't need to tune any hyperparameters here, but you shoud achieve above 40% accuracy after one epoch of training.

In [19]:
```
# We need to wrap `flatten` function in a module in order to stack it
# in nn.Sequential
class Flatten(nn.Module):
    def forward(self, x):
        return flatten(x)

hidden_layer_size = 4000
learning_rate = 1e-2

model = nn.Sequential(
    Flatten(),
    nn.Linear(3 * 32 * 32, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, 10),
)

# you can use Nesterov momentum in optim.SGD
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3589
Checking accuracy on validation set
Got 190 / 1000 correct (19.00)

Iteration 100, loss = 1.7183
Checking accuracy on validation set
Got 401 / 1000 correct (40.10)

Iteration 200, loss = 1.8372
Checking accuracy on validation set
Got 436 / 1000 correct (43.60)

Iteration 300, loss = 1.6652
Checking accuracy on validation set
Got 413 / 1000 correct (41.30)

Iteration 400, loss = 1.8230
Checking accuracy on validation set
Got 409 / 1000 correct (40.90)

Iteration 500, loss = 1.8003
Checking accuracy on validation set
Got 428 / 1000 correct (42.80)

Iteration 600, loss = 1.8922
Checking accuracy on validation set
Got 442 / 1000 correct (44.20)

Iteration 700, loss = 1.9182
Checking accuracy on validation set
Got 460 / 1000 correct (46.00)
```

Sequential API: Three-Layer ConvNet

Here you should use `nn.Sequential` to define and train a three-layer ConvNet with the same architecture we used in Part III:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1

4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You should optimize your model using stochastic gradient descent with Nesterov momentum 0.9.

Again, you don't need to tune any hyperparameters but you should see accuracy above 55% after one epoch of training.

In [20]:
```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
optimizer = None

################################################################################
# TODO: Rewrite the 3-layer ConvNet with bias from Part III with the           #
# Sequential API.                                                              #
################################################################################

model = nn.Sequential(
    nn.Conv2d(3, channel_1, kernel_size=5, padding=2),
    nn.ReLU(),
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2*32*32, 10),
)

optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

# Weight initialization
# Ref: http://pytorch.org/docs/stable/nn.html#torch.nn.Module.apply
def init_weights(m):
    # print(m)
    if type(m) == nn.Conv2d or type(m) == nn.Linear:
        random_weight(m.weight.size())
        zero_weight(m.bias.size())

model.apply(init_weights)

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.2994
Checking accuracy on validation set
Got 121 / 1000 correct (12.10)

Iteration 100, loss = 1.4566
Checking accuracy on validation set
Got 458 / 1000 correct (45.80)

Iteration 200, loss = 1.4757
Checking accuracy on validation set
Got 466 / 1000 correct (46.60)
```

```
Iteration 300, loss = 1.4946
Checking accuracy on validation set
Got 515 / 1000 correct (51.50)

Iteration 400, loss = 1.3858
Checking accuracy on validation set
Got 558 / 1000 correct (55.80)

Iteration 500, loss = 1.0744
Checking accuracy on validation set
Got 561 / 1000 correct (56.10)

Iteration 600, loss = 1.4416
Checking accuracy on validation set
Got 575 / 1000 correct (57.50)

Iteration 700, loss = 1.2354
Checking accuracy on validation set
Got 587 / 1000 correct (58.70)
```

Part V. CIFAR-10 open-ended challenge

In this section, you can experiment with whatever ConvNet architecture you'd like on CIFAR-10.

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves **at least 70%** accuracy on the CIFAR-10 **validation** set within 10 epochs. You can use the check_accuracy and train functions from above. You can use either `nn.Module` or `nn.Sequential` API.

Describe what you did at the end of this notebook.

Here are the official API documentation for each component. One note: what we call in the class "spatial batch norm" is called "BatchNorm2D" in PyTorch.

- Layers in torch.nn package: http://pytorch.org/docs/stable/nn.html
- Activations: http://pytorch.org/docs/stable/nn.html#non-linear-activations
- Loss functions: http://pytorch.org/docs/stable/nn.html#loss-functions
- Optimizers: http://pytorch.org/docs/stable/optim.html

Things you might try:

- **Filter size**: Above we used 5x5; would smaller filters be more efficient?
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?
- **Batch normalization**: Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster?
- **Network architecture**: The network above has two layers of trainable parameters. Can you do better with a deep network? Good architectures to try include:
  - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling**: Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1

image picture (1, 1 , Filter#), which is then reshaped into a (Filter#) vector. This is used in Google's Inception Network (See Table 1 for their architecture).

- **Regularization**: Add l2 weight regularization, or perhaps use Dropout.

Tips for training

For each network architecture that you try, you should tune the learning rate and other hyperparameters. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.
- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are **not required** to implement any of these, but don't miss the fun if you have time!

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.
- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.
- Model ensembles
- Data augmentation
- New Architectures
  - ResNets where the input from the previous layer is added to the output.
  - DenseNets where inputs into previous layers are concatenated together.
  - This blog has an in-depth overview

Have fun and happy training!

In [22]:

```
################################################################################
# TODO:                                                                        #
# Experiment with any architectures, optimizers, and hyperparameters.          #
# Achieve AT LEAST 70% accuracy on the *validation set* within 10 epochs.      #
#                                                                              #
# Note that you can use the check_accuracy function to evaluate on either      #
# the test set or the validation set, by passing either loader_test or         #
# loader_val as the second argument to check_accuracy. You should not touch    #
# the test set until you have finished your architecture and  hyperparameter   #
# tuning, and only run the test set once at the end to report a final value.   #
################################################################################
model = None
optimizer = None

# A 4-layer convolutional network
# (conv -> batchnorm -> relu -> maxpool) * 3 -> fc
layer1 = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=5, padding=2),
    nn.BatchNorm2d(16),
    nn.ReLU(),
```

```
    nn.MaxPool2d(2)
)

layer2 = nn.Sequential(
    nn.Conv2d(16, 32, kernel_size=3, padding=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d(2)
)

layer3 = nn.Sequential(
    nn.Conv2d(32, 64, kernel_size=3, padding=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(2)
)


layer4 = nn.Sequential(
    nn.Linear(64*4*4, 10),
)


model = nn.Sequential(
    layer1,
    layer2,
    layer3,
    Flatten(),
    layer4
)

learning_rate = 1.2e-3

optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Print training status every epoch: set print_every to a large number
print_every = 10000

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

# You should get at least 70% accuracy
train_part34(model, optimizer, epochs=10)
```

```
Iteration 0, loss = 2.8050
Checking accuracy on validation set
Got 119 / 1000 correct (11.90)

Iteration 0, loss = 0.8120
Checking accuracy on validation set
Got 647 / 1000 correct (64.70)

Iteration 0, loss = 0.8660
Checking accuracy on validation set
Got 665 / 1000 correct (66.50)

Iteration 0, loss = 0.9145
Checking accuracy on validation set
Got 688 / 1000 correct (68.80)

Iteration 0, loss = 0.6685
Checking accuracy on validation set
Got 685 / 1000 correct (68.50)
```

```
Iteration 0, loss = 0.7030
Checking accuracy on validation set
Got 695 / 1000 correct (69.50)

Iteration 0, loss = 0.4417
Checking accuracy on validation set
Got 706 / 1000 correct (70.60)

Iteration 0, loss = 0.3442
Checking accuracy on validation set
Got 729 / 1000 correct (72.90)

Iteration 0, loss = 0.5494
Checking accuracy on validation set
Got 755 / 1000 correct (75.50)

Iteration 0, loss = 0.5825
Checking accuracy on validation set
Got 745 / 1000 correct (74.50)
```

Describe what you did

In the cell below you should write an explanation of what you did, any additional features that you implemented, and/or any graphs that you made in the process of training and evaluating your network.

TODO: I managed to increase the validation accuracy to 74.5% and test accuracy to 74.49%. To do this I did not touch the convolutional layers and simply extended the fully connected layers, that is named layer 4. I used a L2 regularization with the weight decay of 1e-6 to decrease the chance of overfitting. However, this affected my accuracy badly. Thus, I have removed it and increased the learning rate instead. I used a ReLU layer to increase complex learning. Furthermore, in the second layer of layer 4, I have also used Softmax as an non-linear activation function. Nonetheless, I did not achieve high precision with neither ReLu nor Softmax. Thus, I have erased them and only used affine layer. It was great practice that will prove beneficial in the final project.

Test set -- run this only once

Now that we've gotten a result we're happy with, we test our final model on the test set (which you should store in best_model). Think about how this compares to your validation set accuracy.

In [23]:
```
best_model = model
check_accuracy_part34(loader_test, best_model)
```

```
Checking accuracy on test set
Got 7449 / 10000 correct (74.49)
```

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import h5py
import seaborn as sn
filename= "assign3_data1.h5"
filename_Q3 = "assign3_data3.h5"
import sys

question = sys.argv[1]

def Atakan_Topcu_21803095_hw3(question):
    if question == '1' :
        ##question 1 code goes here

        file = h5py.File(filename, "r") #Load Data
        # List all groups
        print("Keys: %s" % list(file.keys()))
        data = list(file.keys())[0]
        invXForm = list(file.keys())[1]
        xForm = list(file.keys())[2]

        # Get the data
        data = np.array(file[data])
        invXForm = np.array(file[invXForm])
        xForm = np.array(file[xForm])

        print('Data:', data.shape)
        PreprocessedData=0.2126*data[:,0,:,:]+0.7152*data[:,1,:,:]+0.0722*data[:,
0,:,:]
        print('Preprocessed Data:', PreprocessedData.shape)
        meanIntensity = np.mean(PreprocessedData, axis=(1,2))
        NormalizedData = np.zeros(PreprocessedData.shape)
        #Mean pixel intensity subtraction
        for i in range(PreprocessedData.shape[0]):
                NormalizedData[i,:,:]=PreprocessedData[i,:,:]-meanIntensity[i]
        #Standart deviation
        std = np.std(NormalizedData)
        NormalizedData = np.minimum(NormalizedData, 3*std)
        NormalizedData = np.maximum(NormalizedData, -3*std)
        #[0.1 0.9] scaling
        minRange=0.1
```

```
        maxRange=0.9
        difference=maxRange-minRange
        #Adjusting the scale to the range [0.1 0.9]
        NormalizedData=minRange+((NormalizedData-
np.min(NormalizedData))*difference)/(2*np.max(NormalizedData))
        print(np.min(NormalizedData))
        print(np.max(NormalizedData))

        #Plotting
        row = 10
        col = 20
        fig = plt.figure(figsize=(20, 15), dpi= 160)

        TotalSampleSize = NormalizedData.shape[0]
        randInd = np.random.permutation(TotalSampleSize)

        for i in range(row*col):
                ax = plt.subplot(row, col, i+1)
                plt.imshow(data[randInd[i],:,:,:].T)
                ax.axis('off')
        plt.show()


        fig=plt.figure(figsize=(20, 15), dpi= 160)
        for j in range(row*col):
                ax = plt.subplot(row, col, j+1)
                plt.imshow(NormalizedData[randInd[j],:,:].T, cmap='gray')
                ax.axis('off')
        plt.show()


        #Flatten images to represent each pixel as neuron.
        Pixel = NormalizedData.shape[1]
        SampleSize=NormalizedData.shape[0]
        Flat_Data = np.reshape(NormalizedData, (SampleSize,Pixel**2))
        print("Flattened Data Shape:", Flat_Data.shape)
        Linput=Loutput=Pixel**2
        Lhid = 64
        lmb = 5e-4
        beta = 0.06
        rho = 0.2
        params = (Linput, Lhid, lmb, beta, rho)

        ae=Autoencoder()
        We=ae.WeightInitialization(Linput,Lhid,Loutput)
```

```
We_update=ae.Train(We,Flat_Data,params,0.25,200,640)
# # Part C
row = 8
col = 8
We_final=We_update
fig=plt.figure(figsize=(10, 10))
for i in range(We_final[0].shape[1]):
    plt.subplot(row,col,i+1)
    plt.imshow(np.reshape(We_final[0][:,i],(Pixel,Pixel)), cmap='gray')
    plt.axis('off')
plt.show()




# # Part D

# In[196]:


#Different Hidden Neuron Numbers

Pixel = NormalizedData.shape[1]
SampleSize=NormalizedData.shape[0]
Flat_Data = np.reshape(NormalizedData, (SampleSize,Pixel**2))

Linput=Loutput=Pixel**2
Lhid = 12 #Changed
lmb = 5e-4
beta = 0.06
rho = 0.2
params = (Linput, Lhid, lmb, beta, rho)

ae_12=Autoencoder()
We_12=ae_12.WeightInitialization(Linput,Lhid,Loutput)
We_update_12=ae_12.Train(We_12,Flat_Data,params,0.25,200,640)


# In[197]:


row = 3
col = 4
We_final=We_update_12
fig=plt.figure(figsize=(10, 10))
for i in range(We_final[0].shape[1]):
```

```
        plt.subplot(row,col,i+1)
        plt.imshow(np.reshape(We_final[0][:,i],(Pixel,Pixel)), cmap='gray')
        plt.axis('off')
    plt.show()


    # In[198]:


    #Different Hidden Neuron Numbers

    Pixel = NormalizedData.shape[1]
    SampleSize=NormalizedData.shape[0]
    Flat_Data = np.reshape(NormalizedData, (SampleSize,Pixel**2))

    Linput=Loutput=Pixel**2
    Lhid = 48 #Changed
    lmb = 5e-4
    beta = 0.06
    rho = 0.2
    params = (Linput, Lhid, lmb, beta, rho)

    ae_48=Autoencoder()
    We_48=ae_48.WeightInitialization(Linput,Lhid,Loutput)
    We_update_48=ae_48.Train(We_48,Flat_Data,params,0.25,200,640)


    # In[199]:


    row = 6
    col = 8
    We_final=We_update_48
    fig=plt.figure(figsize=(10, 10))
    for i in range(We_final[0].shape[1]):
        plt.subplot(row,col,i+1)
        plt.imshow(np.reshape(We_final[0][:,i],(Pixel,Pixel)), cmap='gray')
        plt.axis('off')
    plt.show()


    # In[200]:


    #Different Hidden Neuron Numbers
```

```
Pixel = NormalizedData.shape[1]
SampleSize=NormalizedData.shape[0]
Flat_Data = np.reshape(NormalizedData, (SampleSize,Pixel**2))

Linput=Loutput=Pixel**2
Lhid = 96 #Changed
lmb = 5e-4
beta = 0.06
rho = 0.2
params = (Linput, Lhid, lmb, beta, rho)

ae_96=Autoencoder()
We_96=ae_96.WeightInitialization(Linput,Lhid,Loutput)
We_update_96=ae_96.Train(We_96,Flat_Data,params,0.25,200,640)


# In[201]:


row = 8
col = 12
We_final=We_update_96
fig=plt.figure(figsize=(10, 10))
for i in range(We_final[0].shape[1]):
    plt.subplot(row,col,i+1)
    plt.imshow(np.reshape(We_final[0][:,i],(Pixel,Pixel)), cmap='gray')
    plt.axis('off')
plt.show()


# In[11]:


#Different Lamda

Pixel = NormalizedData.shape[1]
SampleSize=NormalizedData.shape[0]
Flat_Data = np.reshape(NormalizedData, (SampleSize,Pixel**2))

Linput=Loutput=Pixel**2
Lhid = 25
lmb = 0 #Changed
beta = 0.06
rho = 0.2
params = (Linput, Lhid, lmb, beta, rho)
```

```
ae_lmb0=Autoencoder()
We_lmb0=ae_lmb0.WeightInitialization(Linput,Lhid,Loutput)
We_update_lmb0=ae_lmb0.Train(We_lmb0,Flat_Data,params,0.25,200,640)


# In[13]:


row = 5
col = 5
We_final=We_update_lmb0
fig=plt.figure(figsize=(10, 10))
for i in range(We_final[0].shape[1]):
    plt.subplot(row,col,i+1)
    plt.imshow(np.reshape(We_final[0][:,i],(Pixel,Pixel)), cmap='gray')
    plt.axis('off')
plt.show()


# In[38]:


#Different Lamda

Pixel = NormalizedData.shape[1]
SampleSize=NormalizedData.shape[0]
Flat_Data = np.reshape(NormalizedData, (SampleSize,Pixel**2))

Linput=Loutput=Pixel**2
Lhid = 25
lmb = 5e-7 #Changed
beta = 0.06
rho = 0.2
params = (Linput, Lhid, lmb, beta, rho)

ae_lmb1=Autoencoder()
We_lmb1=ae_lmb1.WeightInitialization(Linput,Lhid,Loutput)
We_update_lmb1=ae_lmb1.Train(We_lmb1,Flat_Data,params,0.25,200,640)


# In[40]:


row = 5
col = 5
We_final=We_update_lmb1
```

```
        fig=plt.figure(figsize=(10, 10))
        for i in range(We_final[0].shape[1]):
            plt.subplot(row,col,i+1)
            plt.imshow(np.reshape(We_final[0][:,i],(Pixel,Pixel)), cmap='gray')
            plt.axis('off')
        plt.show()


        # In[28]:


        #Different Lamda

        Pixel = NormalizedData.shape[1]
        SampleSize=NormalizedData.shape[0]
        Flat_Data = np.reshape(NormalizedData, (SampleSize,Pixel**2))

        Linput=Loutput=Pixel**2
        Lhid = 25
        lmb = 1e-3 #Changed
        beta = 0.06
        rho = 0.2
        params = (Linput, Lhid, lmb, beta, rho)

        ae_lmb2=Autoencoder()
        We_lmb2=ae_lmb2.WeightInitialization(Linput,Lhid,Loutput)
        We_update_lmb2=ae_lmb2.Train(We_lmb2,Flat_Data,params,0.25,200,640)


        # In[41]:


        row = 5
        col = 5
        We_final=We_update_lmb2
        fig=plt.figure(figsize=(10, 10))
        for i in range(We_final[0].shape[1]):
            plt.subplot(row,col,i+1)
            plt.imshow(np.reshape(We_final[0][:,i],(Pixel,Pixel)), cmap='gray')
            plt.axis('off')
        plt.show()

    elif question == '3' :
        ##question 3 code goes here

        file = h5py.File(filename_Q3, "r") #Load Data
```

```
        # List all groups
        print("Keys: %s" % list(file.keys()))
        trX = list(file.keys())[0]
        trY = list(file.keys())[1]
        tstX = list(file.keys())[2]
        tstY = list(file.keys())[3]

        train_data = np.array(file[trX])
        train_labels = np.array(file[trY])
        test_data = np.array(file[tstX])
        test_labels = np.array(file[tstY])


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

        print("Train Data Size & Train Label Size:",
train_data.shape,train_labels.shape)
        #The length of each time series is 150 units.
        #Labels: (downstairs=1, jogging=2, sitting=3, standing=4, upstairs=5,
walking=6)

        momentum = 0.85
        l_rate = 0.1
        epoch = 30
        batch_size = 32
        RNN_Neuron=128
        indexing=np.random.permutation(train_data.shape[0])
        train_data=train_data[indexing,:,:]
        train_labels=train_labels[indexing,:]

        val_size = int(train_data.shape[0] / 10)
        val_data=train_data[:val_size,:,:]
        val_labels=train_labels[:val_size,:]
        train_data1=train_data[val_size:,:,:]
        train_labels1=train_labels[val_size:,:]

        RNN_Net = RNN_Classifier(train_data1)
        RNN_Net.addLayer(Layer(3, RNN_Neuron,'hyperbolic',1))
        RNN_Net.addLayer(Layer(RNN_Neuron,70,'relu',1))
        RNN_Net.addLayer(Layer(70,30,'relu',1))
        RNN_Net.addLayer(Layer(30,6,'softmax',1))
```

```
        crossList, TrainList =
RNN_Net.TrainNetwork(l_rate,batch_size,train_data1,train_labels1,val_data,val_lab
els,epoch,momentum)

        plt.plot(crossList)
        plt.title('Cross-Entropy Error over Validation Set')
        plt.xlabel('Epoch')
        plt.ylabel('Cross-Entropy Error')
        plt.show()

        TestAcc=RNN_Net.Predict(test_data,test_labels)
        print("Test Accuracy: "+str(TestAcc)+"%")


        TrainAcc=RNN_Net.Predict(train_data1,train_labels1)
        print("Train Accuracy: "+str(TrainAcc)+"%")


        TestConfusion=RNN_Net.ConfusionMatrix(test_data,test_labels)
        names = [1, 2, 3, 4, 5, 6]
        sn.heatmap(TestConfusion, annot=True, annot_kws={"size": 8},
xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
        plt.title("Test Confusion Matrix")
        plt.ylabel("Actual")
        plt.xlabel("Prediction")
        plt.show()


        TrainConfussion=RNN_Net.ConfusionMatrix(train_data1,train_labels1)
        names = [1, 2, 3, 4, 5, 6]
        sn.heatmap(TrainConfussion, annot=True, annot_kws={"size": 8},
xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
        plt.title("Train Confusion Matrix")
        plt.ylabel("Actual")
        plt.xlabel("Prediction")
        plt.show()


        momentum = 0.85
        l_rate = 0.1
        epoch = 30
        batch_size = 32
        LSTM_Neuron=128
        indexing=np.random.permutation(train_data.shape[0])
        train_data=train_data[indexing,:,:]
        train_labels=train_labels[indexing,:]
```

```
        val_size = int(train_data.shape[0] / 10)
        val_data=train_data[:val_size,:,:]
        val_labels=train_labels[:val_size,:]
        train_data1=train_data[val_size:,:,:]
        train_labels1=train_labels[val_size:,:]

        LSTM_Net = LSTM_Classifier(train_data1)
        LSTM_Net.addLayer(LSTM_Layer(131, LSTM_Neuron,1)) #3(Sensor Values) +
128(Previous Values)
        LSTM_Net.addLayer(Layer(LSTM_Neuron,70,'relu',1))
        LSTM_Net.addLayer(Layer(70,30,'relu',1))
        LSTM_Net.addLayer(Layer(30,6,'softmax',1))
        crossList, TrainList =
LSTM_Net.TrainNetwork(l_rate,batch_size,train_data1,train_labels1,val_data,val_la
bels,epoch,momentum)


        plt.plot(crossList)
        plt.title('Cross-Entropy Error over Validation Set')
        plt.xlabel('Epoch')
        plt.ylabel('Cross-Entropy Error')
        plt.show()

        TestAcc=LSTM_Net.Predict(test_data,test_labels)
        print("Test Accuracy: "+str(TestAcc)+"%")


        TrainAcc=LSTM_Net.Predict(train_data1,train_labels1)
        print("Train Accuracy: "+str(TrainAcc)+"%")



        TestConfusion=LSTM_Net.ConfusionMatrix(test_data,test_labels)
        names = [1, 2, 3, 4, 5, 6]
        sn.heatmap(TestConfusion, annot=True, annot_kws={"size": 8},
xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
        plt.title("Test Confusion Matrix")
        plt.ylabel("Actual")
        plt.xlabel("Prediction")
        plt.show()


        TrainConfussion=LSTM_Net.ConfusionMatrix(train_data1,train_labels1)
        names = [1, 2, 3, 4, 5, 6]
```

```
        sn.heatmap(TrainConfussion, annot=True, annot_kws={"size": 8},
xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
        plt.title("Train Confusion Matrix")
        plt.ylabel("Actual")
        plt.xlabel("Prediction")
        plt.show()


        momentum = 0.85
        l_rate = 0.1
        epoch = 30
        batch_size = 32
        GRU_Neuron=128
        indexing=np.random.permutation(train_data.shape[0])
        train_data=train_data[indexing,:,:]
        train_labels=train_labels[indexing,:]

        val_size = int(train_data.shape[0] / 10)
        val_data=train_data[:val_size,:,:]
        val_labels=train_labels[:val_size,:]
        train_data1=train_data[val_size:,:,:]
        train_labels1=train_labels[val_size:,:]

        GRU_Net = GRU_Classifier(train_data1)
        GRU_Net.addLayer(GRU_Layer(3, GRU_Neuron,1)) #3(Sensor Values) +
128(Previous Values)
        GRU_Net.addLayer(Layer(GRU_Neuron,70,'relu',1))
        GRU_Net.addLayer(Layer(70,30,'relu',1))
        GRU_Net.addLayer(Layer(30,6,'softmax',1))
        crossList, TrainList =
GRU_Net.TrainNetwork(l_rate,batch_size,train_data1,train_labels1,val_data,val_lab
els,epoch,momentum)


        plt.plot(crossList)
        plt.title('Cross-Entropy Error over Validation Set')
        plt.xlabel('Epoch')
        plt.ylabel('Cross-Entropy Error')
        plt.show()



        TestAcc=GRU_Net.Predict(test_data,test_labels)
        print("Test Accuracy: "+str(TestAcc)+"%")
```

```
        TrainAcc=GRU_Net.Predict(train_data1,train_labels1)
        print("Train Accuracy: "+str(TrainAcc)+"%")


        # In[24]:


        TestConfusion=GRU_Net.ConfusionMatrix(test_data,test_labels)
        names = [1, 2, 3, 4, 5, 6]
        sn.heatmap(TestConfusion, annot=True, annot_kws={"size": 8},
xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
        plt.title("Test Confusion Matrix")
        plt.ylabel("Actual")
        plt.xlabel("Prediction")
        plt.show()


        # In[25]:


        TrainConfussion=GRU_Net.ConfusionMatrix(train_data1,train_labels1)
        names = [1, 2, 3, 4, 5, 6]
        sn.heatmap(TrainConfussion, annot=True, annot_kws={"size": 8},
xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
        plt.title("Train Confusion Matrix")
        plt.ylabel("Actual")
        plt.xlabel("Prediction")
        plt.show()







#Build Class for RNN and MLP

class Layer: #Layer Class
    def __init__(self,inputDim,numNeurons,activation,beta):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation
        self.beta=beta
        self.w0=np.sqrt(6/(inputDim+numNeurons))
        self.W1=np.random.uniform(-self.w0,self.w0,(inputDim,numNeurons))
        self.b1=np.random.uniform(-self.w0,self.w0,(1, numNeurons))
```

```
        self.weightsAll = np.concatenate((self.W1, self.b1), axis=0)
        self.W2=np.random.uniform(-self.w0,self.w0,(numNeurons,numNeurons))

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

    def activationFunction(self, x):
        if(self.activation == 'hyperbolic'):
            return np.tanh(self.beta*x)
        elif(self.activation == 'softmax'):
            exp_x = np.exp(x - np.max(x))
            return exp_x/np.sum(exp_x, axis=1 ,keepdims=True)
        elif(self.activation=="relu"):
            return np.maximum(x,0)
        elif(self.activation=="sigmoid"):
            exp_x = np.exp(2*x)
            return exp_x/(1+exp_x)
        else:
            return x
    def activationNeuron(self,x):
        x=np.array(x)
        numSamples = x.shape[0]
        tempInp = np.concatenate((x, -1*np.ones((numSamples, 1))), axis=1)
        self.lastActiv =
self.activationFunction(np.matmul(tempInp,self.weightsAll))
        return self.lastActiv

    def RecurrentActivation(self,x,hid):
        x=np.array(x)
        numSamples = x.shape[0]
        tempInp = np.concatenate((x, -1*np.ones((numSamples, 1))), axis=1)
        final=np.matmul(tempInp,self.weightsAll)+np.matmul(hid,self.W2)
        self.lastActiv = self.activationFunction(final)
        return self.lastActiv

    def activation_derivative(self, x):
        if(self.activation == 'hyperbolic'):
            return self.beta*(1-(x**2))
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation=="sigmoid"):
            return (x*(1-x))
```

```
        elif (self.activation=="relu"):
            return 1*(x>0)
        else:
            return np.ones(x.shape)


#Build Class for LSTM
class LSTM_Layer: #LSTM Layer Class
    def __init__(self,inputDim,numNeurons,beta):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.beta=beta
        self.w0=np.sqrt(6/(inputDim+numNeurons))
        # forget gate
        self.W_f=np.random.uniform(-self.w0,self.w0,(inputDim,numNeurons))
        self.bf=np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.Wf = np.concatenate((self.W_f, self.bf), axis=0)
        # input gate
        self.W_i=np.random.uniform(-self.w0,self.w0,(inputDim,numNeurons))
        self.bi=np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.Wi = np.concatenate((self.W_i, self.bi), axis=0)
        # cell gate
        self.W_c=np.random.uniform(-self.w0,self.w0,(inputDim,numNeurons))
        self.bc=np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.Wc = np.concatenate((self.W_c, self.bc), axis=0)
        # output gate
        self.W_o=np.random.uniform(-self.w0,self.w0,(inputDim,numNeurons))
        self.bo=np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.Wo = np.concatenate((self.W_o, self.bo), axis=0)



        self.lastActiv=None
        self.lyrDelta=None
        self.lyrError=None
        self.prevUpdate_f = 0
        self.prevUpdate_i = 0
        self.prevUpdate_c = 0
        self.prevUpdate_o = 0

    def activationFunction(self, x, activation):
        if(activation == 'hyperbolic'):
            return np.tanh(self.beta*x)
        elif(activation == 'softmax'):
            exp_x = np.exp(x - np.max(x))
```

```
            return exp_x/np.sum(exp_x, axis=1)
        elif(activation=="relu"):
            return np.maximum(x,0)
        elif(activation=="sigmoid"):
            exp_x = np.exp(2*x)
            return exp_x/(1+exp_x)
        else:
            return x
    def activationNeuron(self,x, w, activation):
        x=np.array(x)
        numSamples = x.shape[0]
        tempInp = np.concatenate((x, -1*np.ones((numSamples, 1))), axis=1)
        self.lastActiv = self.activationFunction(np.matmul(tempInp,w),activation)
        return self.lastActiv

    def RecurrentActivation(self,x,hid, activation):
        x=np.array(x)
        numSamples = x.shape[0]
        tempInp = np.concatenate((x, -1*np.ones((numSamples, 1))), axis=1)
        final=np.matmul(tempInp,self.weightsAll)+np.matmul(hid,self.W2)
        self.lastActiv = self.activationFunction(final,activation)
        return self.lastActiv

    def activation_derivative(self, x ,activation):
        if(activation == 'hyperbolic'):
            return self.beta*(1-(x**2))
        elif(activation == 'softmax'):
            return x*(1-x)
        elif(activation=="sigmoid"):
            return (x*(1-x))
        elif (activation=="relu"):
            return 1*(x>0)
        else:
            return np.ones(x.shape)


class LSTM_Classifier:
    def __init__(self,training_inputs): #CHECKED
        self.layers=[]
        self.NumSample, self.TimeSample, self.D = training_inputs.shape
        self.Hidden_prev=np.zeros((32, 128))
        self.C_prev=np.zeros((32, 128))

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

```
def FowardProp(self,training_inputs): #CHECKED
    #Foward Propagation
    #LSTM Layer - First Layer
    lyr = self.layers[0]
    N, T, D = training_inputs.shape
    H=128

    z = np.empty((N, T, D + H))
    c = np.empty((N, T, H))
    tanhc = np.empty((N, T, H))
    hf = np.empty((N, T, H))
    hi = np.empty((N, T, H))
    hc = np.empty((N, T, H))
    ho = np.empty((N, T, H))

    h_prev=np.zeros((N, H))
    c_prev=np.zeros((N, H))
    #Though it looks complex, just applying the functions
    for t in range(T):
        z[:, t, :] = np.column_stack((h_prev, training_inputs[:, t, :]))
        zt = z[:, t, :]
        hf[:, t, :] = lyr.activationNeuron(zt , lyr.Wf, "sigmoid")
        hi[:, t, :] = lyr.activationNeuron(zt, lyr.Wi, "sigmoid")
        hc[:, t, :] = lyr.activationNeuron(zt, lyr.Wc, "hyperbolic")
        ho[:, t, :] = lyr.activationNeuron(zt, lyr.Wo, "sigmoid")

        c[:, t, :] = hf[:, t, :] * c_prev + hi[:, t, :] * hc[:, t, :]

        tanhc[:, t, :] = lyr.activationFunction(c[:, t, :], "hyperbolic")

        h_prev = ho[:, t, :] * tanhc[:, t, :]
        c_prev = c[:, t, :]

        cache = {"z_summ": z, #Summation of h_t-1 and x_t
                 "c": c,  #Memory C_t
                 "tanhc": (tanhc), #Tanh of Memory C_t
                 "hf": hf, #Output h_f
                 "hi": (hi), #Output h_i
                 "hc": (hc),#Output h_c
                 "ho": (ho)}#Output h_o


    for layer in self.layers[1:len(self.layers)]: #For MLP Layers
        h_prev=layer.activationNeuron(h_prev)
```

```
        OUT= h_prev
        return cache,OUT

    def
BackProp(self,l_rate,batch_size,training_inputs,training_labels,momentCoef):
        cache,OUT = self.FowardProp(training_inputs)
        foward_out = OUT
        z = cache["z_summ"]
        c=cache["c"]
        tanhc=cache["tanhc"]
        hf=cache["hf"]
        hi=cache["hi"]
        hc=cache["hc"]
        ho=cache["ho"]

        for i in reversed(range(len(self.layers))):# Backpropagation until LSTM
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.lyrDelta=training_labels-foward_out
            elif(lyr==self.layers[0]):
                nextLyr = self.layers[i+1]
                lyr.lyrError = np.matmul(nextLyr.lyrDelta,
nextLyr.weightsAll[0:nextLyr.weightsAll.shape[0]-1,:].T)
                lyr.lyrDelta=lyr.lyrError

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

        dWf = 0
        dWi = 0
        dWc = 0
        dWo = 0
        H=128
        T = z.shape[1]
        NumSample, TimeSample, D = training_inputs.shape
        Prev=np.empty((NumSample, TimeSample, 128))

        lyr0=self.layers[0]
        delta=lyr0.lyrDelta
```

```
        #Backpropagation through time (LSTM)
        for t in reversed(range(T)):
            u = z[:, t, :]
            # if t = 0, c = 0
            if t > 0:
                c_prev = c[:, t - 1, :]
            else:
                c_prev = 0

            dc = delta * ho[:, t, :] * lyr0.activation_derivative(tanhc[:, t,
:],"hyperbolic")
            dhf = dc * c_prev * lyr0.activation_derivative(hf[:, t, :],"sigmoid")
            dhi = dc * hc[:, t, :] * lyr0.activation_derivative(hi[:, t,
:],"sigmoid")
            dhc = dc * hi[:, t, :] * lyr0.activation_derivative(hc[:, t,
:],"sigmoid")
            dho = delta * tanhc[:, t, :] * lyr0.activation_derivative(ho[:, t,
:],"sigmoid")

            dWf += np.matmul(np.concatenate((u, -1*np.ones((NumSample, 1))),
axis=1).T, dhf)
            dWi += np.matmul(np.concatenate((u, -1*np.ones((NumSample, 1))),
axis=1).T, dhi)
            dWc += np.matmul(np.concatenate((u, -1*np.ones((NumSample, 1))),
axis=1).T, dhc)
            dWo += np.matmul(np.concatenate((u, -1*np.ones((NumSample, 1))),
axis=1).T, dho)

            # update the error gradient.
            dxf = np.matmul(dhf, lyr0.Wf.T[:, :H])
            dxi = np.matmul(dhi, lyr0.Wi.T[:, :H])
            dxc = np.matmul(dhc, lyr0.Wc.T[:, :H])
            dxo = np.matmul(dho, lyr0.Wo.T[:, :H])

            delta = (dxf + dxi + dxc + dxo)

        #Updates Weights for first and mlp layers
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):

                update_f =  l_rate*dWf/(batch_size)
                update_i =  l_rate*dWi/(batch_size)
                update_c =  l_rate*dWc/(batch_size)
                update_o =  l_rate*dWo/(batch_size)
```

```
                lyr.Wf+= update_f + (momentCoef*lyr.prevUpdate_f)
                lyr.Wi+= update_i + (momentCoef*lyr.prevUpdate_i)
                lyr.Wc+= update_c + (momentCoef*lyr.prevUpdate_c)
                lyr.Wo+= update_o + (momentCoef*lyr.prevUpdate_o)

                lyr.prevUpdate_f = update_f
                lyr.prevUpdate_i = update_i
                lyr.prevUpdate_c = update_c
                lyr.prevUpdate_o = update_o

            else:
                numSamples = self.layers[i - 1].lastActiv.shape[0]
                tempInp=np.concatenate((self.layers[i - 1].lastActiv, -
1*np.ones((numSamples, 1))), axis=1)
                update =  l_rate*np.matmul(tempInp.T, lyr.lyrDelta)/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):
        crossList = []
        TrainList=[]
        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 = training_inputs[j*batch_size:batch_size*(j+1),:,:]
                train_labels = training_labels[j*batch_size:batch_size*(j+1),:]
                self.BackProp(l_rate,batch_size,train_data,train_labels,momentCoe
f)
            IN, valOutput = self.FowardProp(test_inputs)
            IN1, TrainOutput = self.FowardProp(training_inputs)
            crossErr = np.sum(-np.log(valOutput) *
test_labels)/valOutput.shape[0]
            crossErr1 = np.sum(-np.log(TrainOutput) *
training_labels)/TrainOutput.shape[0]
            print('Cross-Entropy Error for Validation', crossErr)
            print('Cross-Entropy Error for Train', crossErr1)
            crossList.append(crossErr)
            TrainList.append(crossErr1)
```

```
        return crossList, TrainList

    def Predict(self,inputs,output_real):
        Output = self.FowardProp(inputs)[1]
        Output = Output.argmax(axis=1)
        output_real = output_real.argmax(axis=1)
        return ((Output == output_real).mean()*100)
    def ConfusionMatrix(self,input1,output1):
        prediction= self.FowardProp(input1)[1]
        prediction = prediction.argmax(axis=1)
        output1 = output1.argmax(axis=1)
        K = len(np.unique(output1))
        c=np.zeros((K,K))
        for i in range(len(output1)):
            c[output1[i]][prediction[i]] += 1
        return c


#Build Class for GRU
class GRU_Layer: #GRU Layer Class
    def __init__(self,inputDim,numNeurons,beta):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.beta = beta
        self.w0=np.sqrt(6/(inputDim+numNeurons))
        self.w1=np.sqrt(6/(numNeurons+numNeurons))

        self.W_z = np.random.uniform(-self.w0, self.w0,
size=(inputDim,numNeurons))
        self.bz = np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.Uz = np.random.uniform(-self.w1, self.w1, size=(numNeurons,
numNeurons))
        self.Wz = np.concatenate((self.W_z, self.bz), axis=0)

        self.W_r = np.random.uniform(-self.w0, self.w0,
size=(inputDim,numNeurons))
        self.br = np.random.uniform(-self.w0,self.w0,(1, numNeurons))
        self.Ur = np.random.uniform(-self.w1, self.w1, size=(numNeurons,
numNeurons))
        self.Wr = np.concatenate((self.W_r, self.br), axis=0)

        self.W_h = np.random.uniform(-self.w0, self.w0,
size=(inputDim,numNeurons))
        self.bh = np.random.uniform(-self.w0,self.w0,(1, numNeurons))
```

```
        self.Uh = np.random.uniform(-self.w1, self.w1, size=(numNeurons,
numNeurons))
        self.Wh = np.concatenate((self.W_h, self.bh), axis=0)


        self.lastActiv=None
        self.lyrDelta=None
        self.lyrError=None
        self.prevUpdate_Wz = 0
        self.prevUpdate_Wr = 0
        self.prevUpdate_Wh = 0

        self.prevUpdate_Uz = 0
        self.prevUpdate_Ur = 0
        self.prevUpdate_Uh = 0

    def activationFunction(self, x, activation):
        if(activation == 'hyperbolic'):
            return np.tanh(self.beta*x)
        elif(activation == 'softmax'):
            exp_x = np.exp(x - np.max(x))
            return exp_x/np.sum(exp_x, axis=1)
        elif(activation=="relu"):
            return np.maximum(x,0)
        elif(activation=="sigmoid"):
            exp_x = np.exp(2*x)
            return exp_x/(1+exp_x)
        else:
            return x
    def activationNeuron(self,x, w, h, u, activation):
        x=np.array(x)
        numSamples = x.shape[0]
        tempInp = np.concatenate((x, -1*np.ones((numSamples, 1))), axis=1)
        self.lastActiv =
self.activationFunction(np.matmul(tempInp,w)+np.matmul(h,u),activation)
        return self.lastActiv

    def activation_derivative(self, x ,activation):
        if(activation == 'hyperbolic'):
            return self.beta*(1-(x**2))
        elif(activation == 'softmax'):
            return x*(1-x)
        elif(activation=="sigmoid"):
            return (x*(1-x))
        elif (activation=="relu"):
```

```
            return 1*(x>0)
        else:
            return np.ones(x.shape)


class GRU_Classifier:
    def __init__(self,training_inputs): #CHECKED
        self.layers=[]
        self.NumSample, self.TimeSample, self.D = training_inputs.shape
        self.Hidden_prev=np.zeros((32, 128))
        self.C_prev=np.zeros((32, 128))

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

    def FowardProp(self,training_inputs): #CHECKED
        #Foward Propagation
        #GRU Layer - First Layer
        lyr = self.layers[0]
        N, T, D = training_inputs.shape
        H=128

        z = np.empty((N, T, H))
        r = np.empty((N, T, H))
        h_tilde = np.empty((N, T, H))
        h = np.empty((N, T, H))
        h_prev=np.zeros((N, H))
        #Similar to LSTM function. Just applying the functions.
        for t in range(T):
            x = training_inputs[:, t, :]
            z[:, t, :] = lyr.activationNeuron(x,  lyr.Wz, h_prev, lyr.Uz ,
"sigmoid")
            r[:, t, :] = lyr.activationNeuron(x, lyr.Wr, h_prev,  lyr.Ur ,
"sigmoid")
            h_tilde[:, t, :] = lyr.activationNeuron(x, lyr.Wh, (r[:, t, :] *
h_prev), lyr.Uh, "hyperbolic")
            h[:, t, :] = (1 - z[:, t, :]) * h_prev + z[:, t, :] * h_tilde[:, t,
:]
            h_prev = h[:, t, :]

            cache = {"z": z,
                     "r": r,
                     "h_tilde": (h_tilde),
                     "h": h}
```

```
        for layer in self.layers[1:len(self.layers)]: #For MLP Layers
            h_prev=layer.activationNeuron(h_prev)
        OUT= h_prev
        return cache,OUT

    def
BackProp(self,l_rate,batch_size,training_inputs,training_labels,momentCoef):
        cache,OUT = self.FowardProp(training_inputs)
        foward_out = OUT
        z = cache["z"]
        r=cache["r"]
        h_tilde=cache["h_tilde"]
        h=cache["h"]


        for i in reversed(range(len(self.layers))):# Backpropagation until LSTM
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.lyrDelta=training_labels-foward_out
            elif(lyr==self.layers[0]):
                nextLyr = self.layers[i+1]
                lyr.lyrError = np.matmul(nextLyr.lyrDelta,
nextLyr.weightsAll[0:nextLyr.weightsAll.shape[0]-1,:].T)
                lyr.lyrDelta=lyr.lyrError

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

        dWr = 0
        dUr = 0

        dWh = 0
        dUh = 0

        H=128
        NumSample, T, D = training_inputs.shape
```

```
        Prev=np.empty((NumSample, T, 128))

        lyr0=self.layers[0]
        delta=lyr0.lyrDelta
        #Backpropagation through time (GRU)
        for t in reversed(range(T)):
            x = training_inputs[:, t, :]
            # if t = 0, h_prev = 0, similar to LSTM
            if t > 0:
                h_prev = h[:, t - 1, :]
            else:
                h_prev = np.zeros((NumSample, H))

            # dE/dr is named as dr which is true for all variables (dz, dh_tilde)
            dz = delta * (h_tilde[:, t, :] - h_prev) *
lyr0.activation_derivative(z[:, t, :],"sigmoid")
            dh_tilde = delta * z[:, t, :] * lyr0.activation_derivative(h_tilde[:,
t, :],"hyperbolic")
            dr = (np.matmul(dh_tilde, lyr0.Uh.T) * h_prev *
lyr0.activation_derivative(r[:, t, :],"sigmoid"))

            dWz += np.matmul(np.concatenate((x, -1*np.ones((NumSample, 1))),
axis=1).T, dz)
            dUz += np.matmul(h_prev.T, dz)

            dWr += np.matmul(np.concatenate((x, -1*np.ones((NumSample, 1))),
axis=1).T, dr)
            dUr += np.matmul(h_prev.T, dr)

            dWh += np.matmul(np.concatenate((x, -1*np.ones((NumSample, 1))),
axis=1).T, dh_tilde)
            dUh += np.matmul(h_prev.T, dh_tilde)

            # update the error gradient.


            d1 = delta * (1 - z[:, t, :])
            d2 = np.matmul(dz, lyr0.Uz.T)
            d3 = np.matmul(dh_tilde, lyr0.Uh.T) * (r[:, t, :] + h_prev *
np.matmul(lyr0.activation_derivative(r[:, t, :],"sigmoid"),
lyr0.Ur.T))

            delta = (d1+d2+d3)

        #Updates Weights for first and mlp layers
```

```
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):

                update_Wz =  l_rate*dWz/(batch_size)
                update_Wr =  l_rate*dWr/(batch_size)
                update_Wh =  l_rate*dWh/(batch_size)
                update_Uz =  l_rate*dUz/(batch_size)
                update_Ur =  l_rate*dUr/(batch_size)
                update_Uh =  l_rate*dUh/(batch_size)




                lyr.Wz+= update_Wz + (momentCoef*lyr.prevUpdate_Wz)
                lyr.Wr+= update_Wr + (momentCoef*lyr.prevUpdate_Wr)
                lyr.Wh+= update_Wh + (momentCoef*lyr.prevUpdate_Wh)
                lyr.Uz+= update_Uz + (momentCoef*lyr.prevUpdate_Uz)
                lyr.Ur+= update_Ur + (momentCoef*lyr.prevUpdate_Ur)
                lyr.Uh+= update_Uh +
(momentCoef*lyr.prevUpdate_Uh)


                lyr.prevUpdate_Wz = update_Wz
                lyr.prevUpdate_Wr = update_Wr
                lyr.prevUpdate_Wh = update_Wh

                lyr.prevUpdate_Uz = update_Uz
                lyr.prevUpdate_Ur = update_Ur
                lyr.prevUpdate_Uh = update_Uh


            else:
                numSamples = self.layers[i - 1].lastActiv.shape[0]
                tempInp=np.concatenate((self.layers[i - 1].lastActiv, -
1*np.ones((numSamples, 1))), axis=1)
                update =  l_rate*np.matmul(tempInp.T, lyr.lyrDelta)/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):
        crossList = []
        TrainList=[]
        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 = training_inputs[j*batch_size:batch_size*(j+1),:,:]
                train_labels = training_labels[j*batch_size:batch_size*(j+1),:]
                self.BackProp(l_rate,batch_size,train_data,train_labels,momentCoe
f)
            IN, valOutput = self.FowardProp(test_inputs)
            IN1, TrainOutput = self.FowardProp(training_inputs)
            crossErr = np.sum(-np.log(valOutput) *
test_labels)/valOutput.shape[0]
            crossErr1 = np.sum(-np.log(TrainOutput) *
training_labels)/TrainOutput.shape[0]
            print('Cross-Entropy Error for Validation', crossErr)
            print('Cross-Entropy Error for Train', crossErr1)
            crossList.append(crossErr)
            TrainList.append(crossErr1)
        return crossList, TrainList

    def Predict(self,inputs,output_real):
        Output = self.FowardProp(inputs)[1]
        Output = Output.argmax(axis=1)
        output_real = output_real.argmax(axis=1)
        return ((Output == output_real).mean()*100)
    def ConfusionMatrix(self,input1,output1):
        prediction= self.FowardProp(input1)[1]
        prediction = prediction.argmax(axis=1)
        output1 = output1.argmax(axis=1)
        K = len(np.unique(output1))
        c=np.zeros((K,K))
        for i in range(len(output1)):
            c[output1[i]][prediction[i]] += 1
        return c



class RNN_Classifier:
    def __init__(self,training_inputs):
        self.layers=[]
        self.NumSample, self.TimeSample, self.D = training_inputs.shape
        self.Hidden_prev=np.zeros((32, 128))
```

```
        self.RecurrentError=np.empty((32, self.TimeSample, 128))
        self.RecurrentDelta=np.empty((32, self.TimeSample, 128))

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

    def FowardProp(self,training_inputs):
        #Foward Propagation

        #RNN Layer - First Layer
        NumSample, TimeSample, D = training_inputs.shape
        IN=np.empty((NumSample, TimeSample, 128))
        self.Hidden_prev=np.zeros((NumSample, 128))
        for time in range(TimeSample): #You have to take into account whole time
samples.
            x = training_inputs[:, time, :]
            IN[:, time, :]=self.layers[0].RecurrentActivation(x,self.Hidden_prev)
            self.Hidden_prev=IN[:, time, :]
        OUT = IN[:, -1, :]

        for layer in self.layers[1:len(self.layers)]: #For MLP Layers
            OUT=layer.activationNeuron(OUT) #Only use the last time sample as it
contains memory.
        return IN,OUT

    def
BackProp(self,l_rate,batch_size,training_inputs,training_labels,momentCoef):
        IN,OUT = self.FowardProp(training_inputs)
        foward_out = OUT
        for i in reversed(range(len(self.layers))):# Backpropagation until
recurrent
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.lyrDelta=training_labels-foward_out
            elif (lyr == self.layers[0]):
                nextLyr = self.layers[i+1]
                lyr.lyrError = np.matmul(nextLyr.lyrDelta,
nextLyr.weightsAll[0:nextLyr.weightsAll.shape[0]-1,:].T)
                self.RecurrentError[:,-1,:]=lyr.lyrError
                derivative=lyr.activation_derivative(lyr.lastActiv)
                lyr.lyrDelta=derivative*lyr.lyrError
                self.RecurrentDelta[:,-1,:]=lyr.lyrDelta
            else:
                nextLyr = self.layers[i+1]
```

```
                lyr.lyrError = np.matmul(nextLyr.lyrDelta,
nextLyr.weightsAll[0:nextLyr.weightsAll.shape[0]-1,:].T)
                derivative=lyr.activation_derivative(lyr.lastActiv)
                lyr.lyrDelta=derivative*lyr.lyrError
        dWall=0
        dWhid=0
        NumSample, TimeSample, D = training_inputs.shape
        Prev=np.empty((NumSample, TimeSample, 128))
        #Backpropagation through time (Recurrent)
        for time in reversed(range(TimeSample)):
            lyr=self.layers[0]
            #u = IN[:, time-1, :]
            x=training_inputs[:,time,:]

            if time > 0:
                u = IN[:, time-1, :]
            else:
                u = np.zeros((NumSample, 128))

            derivative=lyr.activation_derivative(u)
            dWhid+=np.matmul(u.T, self.RecurrentDelta[:,time,:])
            dWall+=np.matmul(np.concatenate((training_inputs[:,time-1,:], -
1*np.ones((training_inputs[:,time-1,:].shape[0], 1))), axis=1) .T,
self.RecurrentDelta[:,time,:])

            #For Recurrent Delta Update
            self.RecurrentError[:,time-
1,:]=np.matmul(self.RecurrentDelta[:,time,:],lyr.W2.T)
            self.RecurrentDelta[:,time-
1,:]=derivative*self.RecurrentError[:,time-1,:]

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                update_1 =  l_rate*dWall/(batch_size*150)
                update_2 =  l_rate*dWhid/(batch_size*150)
                lyr.weightsAll+= update_1 + (momentCoef*lyr.prevUpdate)
                lyr.W2+=update_2 + (momentCoef*lyr.prevUpdateRNN)
                lyr.prevUpdate = update_1
                lyr.prevUpdateRNN = update_2

            else:
                numSamples = self.layers[i - 1].lastActiv.shape[0]
```

```
                tempInp=np.concatenate((self.layers[i - 1].lastActiv, -
1*np.ones((numSamples, 1))), axis=1)
                update =  l_rate*np.matmul(tempInp.T, lyr.lyrDelta)/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):
        crossList = []
        TrainList=[]
        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 = training_inputs[j*batch_size:batch_size*(j+1),:,:]
                train_labels = training_labels[j*batch_size:batch_size*(j+1),:]
                self.BackProp(l_rate,batch_size,train_data,train_labels,momentCoe
f)
            IN, valOutput = self.FowardProp(test_inputs)
            IN1, TrainOutput = self.FowardProp(training_inputs)
            crossErr =  np.sum(-np.log(valOutput) *
test_labels)/valOutput.shape[0]
            crossErr1 =  np.sum(-np.log(TrainOutput) *
training_labels)/TrainOutput.shape[0]
            print('Cross-Entropy Error for Validation', crossErr)
            print('Cross-Entropy Error for Train', crossErr1)
            crossList.append(crossErr)
            TrainList.append(crossErr1)
        return crossList, TrainList

    def Predict(self,inputs,output_real):
        Output = self.FowardProp(inputs)[1]
        Output = Output.argmax(axis=1)
        output_real = output_real.argmax(axis=1)
        return ((Output == output_real).mean()*100)

    def ConfusionMatrix(self,input1,output1):
        prediction= self.FowardProp(input1)[1]
        prediction = prediction.argmax(axis=1)
        output1 = output1.argmax(axis=1)
        K = len(np.unique(output1))
```

```
        c=np.zeros((K,K))
        for i in range(len(output1)):
            c[output1[i]][prediction[i]] += 1
        return c

class Autoencoder():
    def w0(self,Lpre,Lpost): #Lpre;post are the number of neurons on either side
of the connection weights.
        return np.sqrt(6/(Lpre+Lpost))

    def WeightInitialization(self, Lin, Lhidden, Lout):
        np.random.seed(99)
        W1=np.random.uniform(-self.w0(Lin, Lhidden),self.w0(Lin,
Lhidden),(Lin,Lhidden))
        W2=np.random.uniform(-self.w0(Lhidden, Lout),self.w0(Lhidden,
Lout),(Lhidden, Lout))
        b1=np.random.uniform(-self.w0(Lin, Lhidden),self.w0(Lin, Lhidden),(1,
Lhidden))
        b2=np.random.uniform(-self.w0(Lhidden, Lout),self.w0(Lhidden, Lout),(1,
Lout))
        We=(W1, W2, b1, b2)
        return We

    def sigmoid(self,x):
        #Sigmoid Function
        expx = np.exp(x)
        return expx/(1+expx)

    def sigmoidDerivative(self,x):
        return x*(1-x)

    def fowardpass(self,We,data): #Simple Foward Pass
        W1, W2, b1, b2 = We
        W1 = np.concatenate((W1,b1), axis=0)
        W2 = np.concatenate((W2,b2), axis=0)
        data_new = np.concatenate((data, np.ones((data.shape[0], 1))), axis=1)
        hid = self.sigmoid(np.matmul(data_new,W1))
        hid_new = np.concatenate((hid, np.ones((hid.shape[0], 1))), axis=1)
        out = self.sigmoid(np.matmul(hid_new,W2))
        return hid, out

    def aeCost(self, We, data, params):
        (Lin, Lhid, lmb, beta, rho) = params #Extra Parameters
        W1, W2, b1, b2 = We #Weights
        N = data.shape[0] #Sample Size
```

```
        hidden, dataResult = self.fowardpass(We, data)
        hidden_mean = np.mean(hidden, axis=0)

        MSE = (1/(2*N))*np.sum(np.power((data - dataResult),2))
        Tyk = (lmb/2)*(np.sum(W1**2) + np.sum(W2**2))
        kl1 = rho*np.log(hidden_mean/rho)
        kl2 = (1-rho)*np.log((1-hidden_mean)/(1-rho))
        KL_final =  beta*np.sum(kl1+kl2)

        J = MSE + Tyk + KL_final

        deltaOut = -(data-dataResult)*self.sigmoidDerivative(dataResult)
        derKL = np.tile(beta*(-(rho/hidden_mean.T)+((1-rho)/(1-hidden_mean.T))),
(N,1)).T
        deltaHid = (np.matmul(W2,deltaOut.T)+ derKL) *
self.sigmoidDerivative(hidden).T

        gradWout = (1/N)*(np.matmul(deltaOut.T,hidden).T + lmb*W2)
        gradBout = np.mean(deltaOut, axis=0)
        gradWhid = (1/N)*(np.matmul(data.T,deltaHid.T) + lmb*W1)
        gradBhid = np.mean(deltaHid, axis=1)

        Jgrad=(gradWhid, gradWout, gradBhid, gradBout)

        return J, Jgrad

    def update(self, We, data, params,l_rate):
        (Lin, Lhid, lmb, beta, rho) = params #Extra Parameters
        W1, W2, b1, b2 = We #Weights
        N = data.shape[0] #Sample Size
        J, Jgrad = self.aeCost(We, data, params)
        W1 = W1 - Jgrad[0]*l_rate
        W2 = W2 - Jgrad[1]*l_rate
        b1 = b1 - Jgrad[2]*l_rate
        b2 = b2 - Jgrad[3]*l_rate
        newWeights=(W1, W2, b1, b2)
        return J,newWeights

    def Train(self, We, data, params,l_rate, epochs, batch_size):
        for epoch in range(epochs):
            indexing=np.random.permutation(data.shape[0])
            data=data[indexing,:]
            numBatches = int(np.floor(data.shape[0]/batch_size))
            Total_Loss=0
```

```
            for j in range(numBatches):
                loss, We =
self.update(We,data[j*batch_size:batch_size*(j+1),:],params,l_rate)
                Total_Loss=Total_Loss+loss

            Total_Loss=Total_Loss/numBatches
            print('Epoch', epoch+1)
            print('Loss', Total_Loss)
        return We

Atakan_Topcu_21803095_hw3(question)
```