EEE443 - NEURAL NETWORKS

HOMEWORK ASSIGNMENT - 3

Berkan Ozdamar
21602353

Question 1

In this question we are asked to implement an auto encoder network with a single layer for unsupervised feature extraction. The network's loss function is given below as:

$$J_{ae} = \frac{1}{2N}\sum_{i=1}^{N}\|d(m)-o(m)\|^2 + \frac{\lambda}{2}\left[\sum_{b=1}^{L_{hid}}\sum_{a=1}^{L_{in}}(W_{a,b}^{(1)})^2 + \sum_{c=1}^{L_{out}}\sum_{b=1}^{L_{hid}}(W_{b,c}^{(2)})^2\right] + \beta\sum_{b=1}^{L_{hid}} KL(\rho|\hat{\rho}_b)$$

In the cost function, the first term is the average squared-error between the input and the network output. The second terms is the Tykhonoc regularization and the third term is the Kullback-Leibler divergence term. Kullback-Leibler divergence sparsity is controlled with the term ρ.

Part a

In this part, the 10240 images are transferred to grayscale using the Luminosity model which is given as:

$$Y = 0.02126 + R + 0.7152 + G + 0.0722 + B$$

where R represent the red channel of each image, G is the green channel of each image and finally B is the blue channel of each image. Then the images are normalized by subtracting the mean pixel intensity of each image from themselves. Next, the data is clipped to ±3 standard deviations measured accross the images. Finally, the data is mapped to range [0.1, 0.9].

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

[2]:
```
# Part A

f = h5py.File('assign3_data1.h5', 'r')
dataKeys = list(f.keys())
print('The data keys are:' + str(dataKeys))

# Gathering the  train images, test images, train labels and test labels.
data = f['data']
invXForm = f['invXForm']
xForm = f['xForm']

# data=np.array(data)
# invXForm=np.array(invXForm)
# xForm=np.array(xForm)

# data = data.reshape(-1,16,16,3)

print('The size of data is: ' + str(np.shape(data)))
print('The size of invXForm is: ' + str(np.shape(invXForm)))
print('The size of xForm is: ' + str(np.shape(xForm)))
```

```
The data keys are:['data', 'invXForm', 'xForm']
The size of data is: (10240, 3, 16, 16)
The size of invXForm is: (105, 768)
The size of xForm is: (768, 105)
```

[3]:
```
data_r = data[:,0,:,:]
data_g = data[:,1,:,:]
data_b = data[:,2,:,:]

data_grayscale = data_r*0.2126 + data_g*0.7152 + data_b*0.0722
```

```
print(np.shape(data_grayscale))
```

```
(10240, 16, 16)
```

[4]:
```
def normalize_data(images):
    data_mean = np.mean(images, axis=(1,2))
    for i in range(np.shape(data_mean)[0]):
        images[i,:,:] -= data_mean[i]
    return images
```

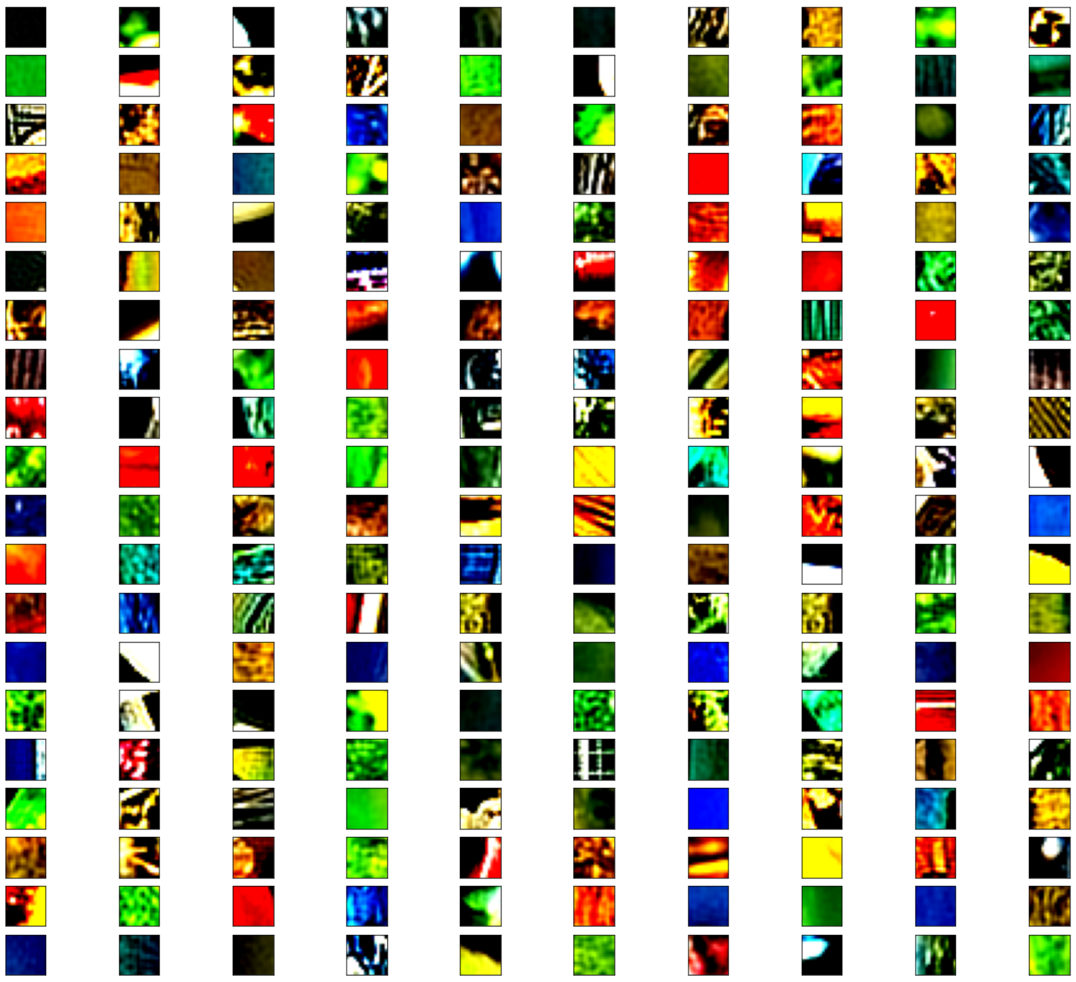
[5]:
```
def map_std(images):
    data_std = np.std(images)
    mapped_data = np.where(images > 3*data_std, 3*data_std, images)
    mapped_data_final = np.where(mapped_data < -3*data_std, -3*data_std,␣
 ↪mapped_data)
    return mapped_data_final
```

[6]:
```
def clip_data_range(images, min_value, max_value):
    range_val = max_value - min_value
    max_data = np.max(images)
    min_data = np.min(images)

    result = images - min_data

    max_data = np.max(result)
    result = result / max_data * range_val

    result = result + min_value
    return result
```

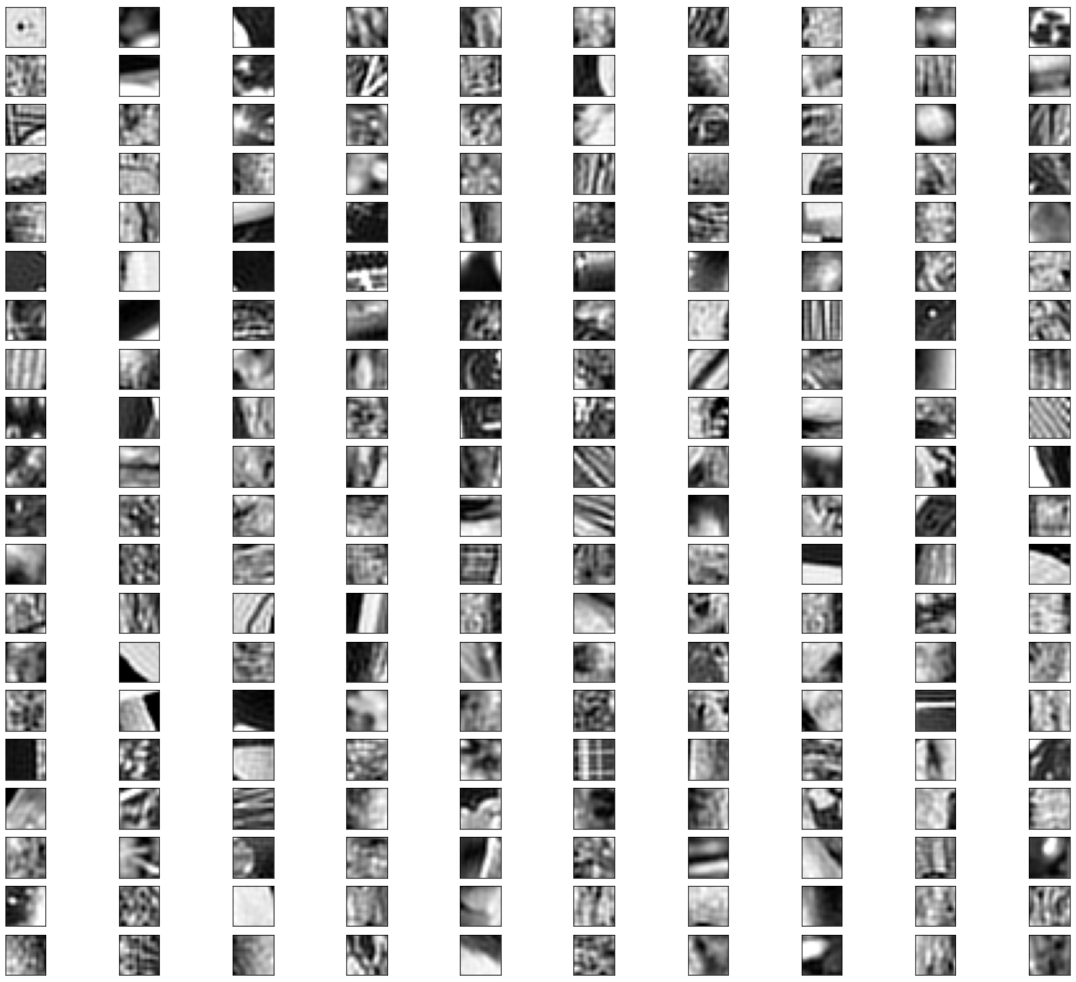
[7]:
```
data_grayscale_norm = normalize_data(data_grayscale)
data_grayscale_norm_mapped = map_std(data_grayscale_norm)
data_final = clip_data_range(data_grayscale_norm_mapped, 0.1, 0.9)
```

Then the random 200 images and the manipulated images are visualized respectively below as:

[8]:
```
figureNum = 0
plt.figure(figureNum, figsize=(18, 16))
np.random.seed(9)
sample_size = np.shape(data_final)[0]
random_200 = np.random.randint(sample_size, size=(200))

for i,value in enumerate(random_200):
    ax1 = plt.subplot(20, 10,i+1)
    ax1.imshow(np.transpose(data[value], (1,2,0)))
    ax1.set_yticks([])
    ax1.set_xticks([])

plt.show()
```



[9]:
```
figureNum += 1
plt.figure(figureNum, figsize=(18, 16))

for subplot,value in enumerate(random_200):
    ax2 = plt.subplot(20, 10,subplot+1)
    ax2.imshow(data_final[value], cmap='gray')
    ax2.set_yticks([])
    ax2.set_xticks([])
plt.show()
```



Here, it can be seen that the manipulated images resemble the original images, However, some images do not resemble the original images since the the original images are clipped but mostly the manipulated data resemples the original data.

Part b

In this part, we are implementing the autoencodes with the cost function discussed previously. The weights will initialized uniformly distributed in the interval [-lim, lim] where the lim is:

$$lim = \sqrt{\frac{6}{L_{pre} + L_{post}}}$$

Then, the cost function aeCost is implemented which calculates the partial derivatives of weights and bias terms. The $L_{hid} = 64$. Since each image has dimension of (16,16), the shapes of $L_{pre} = L_{post} = 16 * 16 = 256$. The parameters are specified as $\lambda = 5x10^{-4}$ and β and ρ values will be tuned. For the training of the model, mini-batch gradient descent is implemented.

The implementation of the model is given below as:

[10]:
```
# Part B

def sigmoid(x):

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

def der_sigmoid(x):

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


def forward(We, data):

    W1, B1, W2, B2 = We

    # HIDDEN LAYER
    A1 = data.dot(W1) + B1
    Z1 = sigmoid(A1)
    # OUTPUT LAYER
    A2 = Z1.dot(W2) + B2
    y_pred = sigmoid(A2)


    return A1, Z1, A2, y_pred


def aeCost(We, data, params):
    Lin, Lhid, lambdaa, beta, rho = params
    W1, B1, W2, B2  = We
    sample_size = np.shape(data)[0]

    A1, Z1, A2, y_pred = forward(We, data)
    Z1_mean = np.mean(Z1, axis=0)

    J_1 = (1/(2*sample_size))*np.sum(np.power((data - y_pred),2))
    J_2 = (lambdaa / 2)* (np.sum(W1**2) + np.sum(W2**2))
    KL_1 = rho*np.log(Z1_mean/rho)
    KL_2 = (1-rho)*np.log((1-Z1_mean)/(1-rho))
    J_3 =  beta*np.sum(KL_1+KL_2)
    J = J_1 + J_2 - J_3

    del_out = -(data - y_pred) * der_sigmoid(y_pred)

    del_KL = beta*(-(rho/Z1_mean.T)+((1-rho)/(1-Z1_mean.T)))
    del_KLs =  np.vstack([del_KL]*sample_size)

    del_hidden = ((del_out.dot(W1)) + del_KLs) * der_sigmoid(Z1)

    # Gradients
    grad_W2 = (1/ sample_size)*(Z1.T.dot(del_out) + lambdaa*W2)
    grad_B2 = np.mean(del_out, axis=0, keepdims = True)

    grad_W1 = (1/ sample_size)*(data.T.dot(del_hidden) + lambdaa*W1)
    grad_B1 = np.mean(del_hidden, axis=0, keepdims = True)

    gradients = [grad_W2, grad_B2, grad_W1, grad_B1]

    return J, gradients
```

```python
def update_weights(We, data, params, learning_rate):

    J, gradients = aeCost(We, data, params)
    grad_W2, grad_B2, grad_W1, grad_B1 = gradients
    W1, B1, W2, B2 = We

    # Update weights

    W2  -= learning_rate * grad_W2
    B2 -= learning_rate * grad_B2

    W1  -= learning_rate * grad_W1
    B1 -= learning_rate * grad_B1


    We_updated = [W1, B1, W2, B2]
    return J, We_updated


def initialize_weights(Lpre, Lhid):

    np.random.seed(8)

    Lpost = Lpre
    lim_1 = np.sqrt(6/(Lpre+Lhid))
    lim_2 = np.sqrt(6/(Lhid+Lpost))

    W1 = np.random.uniform(-lim_1, lim_1, (Lpre,Lhid))
    B1 = np.random.uniform(-lim_1, lim_1, (1,Lhid))

    W2 = np.random.uniform(-lim_2, lim_2, (Lhid,Lpost))
    B2 = np.random.uniform(-lim_2, lim_2, (1,Lpost))

    return W1, B1, W2, B2


def train_network(data, params, learning_rate, batch_size, epoch):

    np.random.seed(8)

    sample_size = np.shape(data)[0]
    Lin, Lhid, lambdaa, beta, rho = params
    W1, B1, W2, B2  = initialize_weights(Lin, Lhid)

    We = [W1, B1, W2, B2]
    Loss = list()
    for i in range(epoch):
        if(i % 10 == 0):
            print('Epoch: ' + str(i))
        # Randomize the dataset for each iteration
        randomIndexes = np.random.permutation(sample_size)
        data = data[randomIndexes]

        number_of_batches = int(sample_size / batch_size)

        for j in range(number_of_batches):
            # Mini batch start and end index
            start = int(batch_size*j)
            end = int(batch_size*(j+1))

            _, We = update_weights(We, data[start:end], params, learning_rate)

        J,_ = aeCost(We, data, params)
        Loss.append(J)

    return Loss, We
```

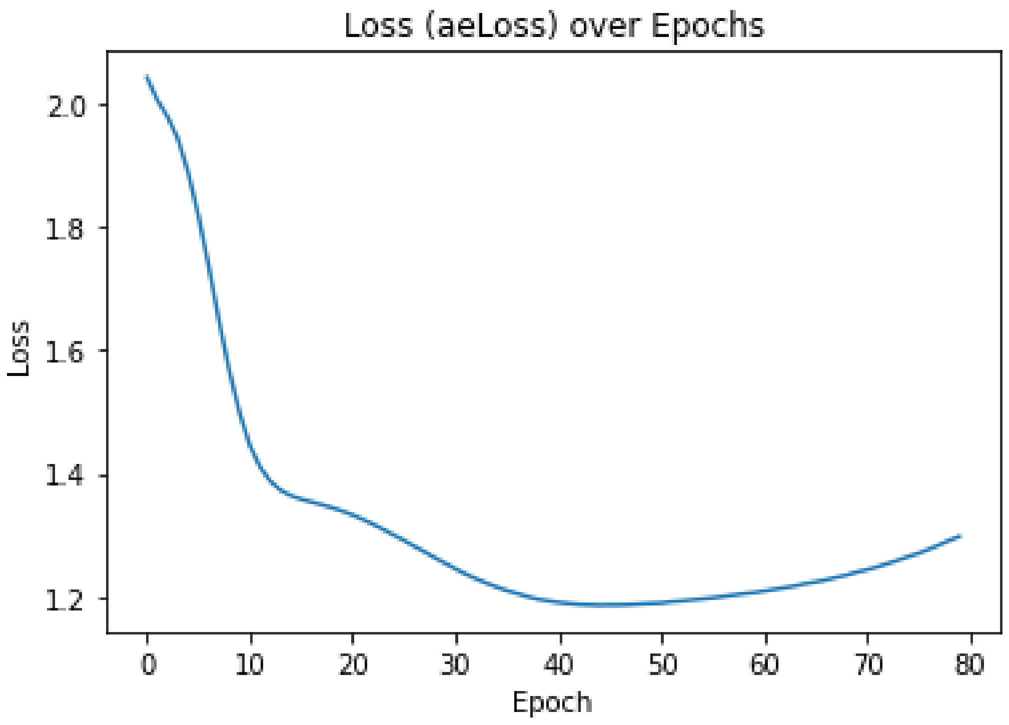
The tuned hyperparameters are $\beta = 0.01$, $\rho = 0.2$ and the learning rate = 0.01.

[11]:
```
data_final_flat = np.reshape(data_final, (np.shape(data_final)[0], 16**2))

Lin = Lpost = 16**2
Lhid = 64
lambdaa = 5e-4
beta = 0.01
rho = 0.2
params = [Lin, Lhid, lambdaa, beta, rho]
```

[12]:
```
loss, We_t = train_network(data_final_flat, params, 1e-2, 16, 80)
```

The loss over images over 80 epochs is given below:

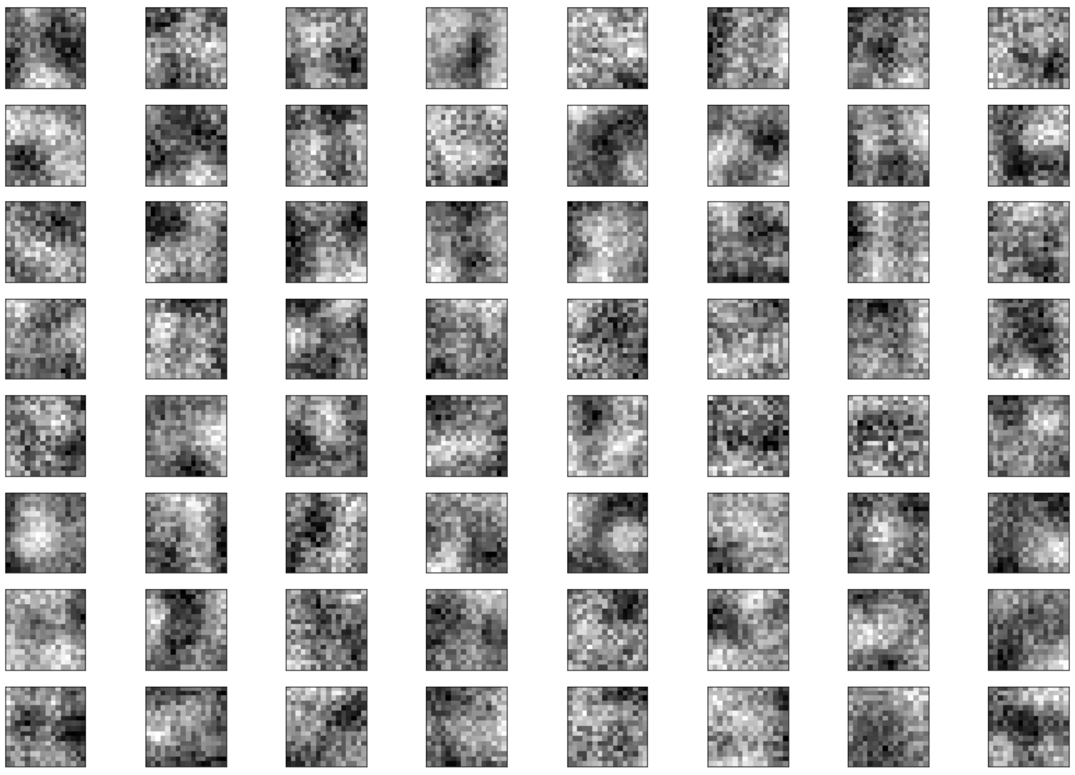
[13]:
```
figureNum += 1
plt.figure(figureNum)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss (aeLoss) over Epochs')
plt.plot(loss)
plt.show()
```



Part C

In this part, we are asked to display the first layer of connection weights as a separate image for each neuron in the hidden layer. Then the weights resembles with the images will be discussed. The first layer of connection weights as a separate image for each neuron in the hidden layer is given below:

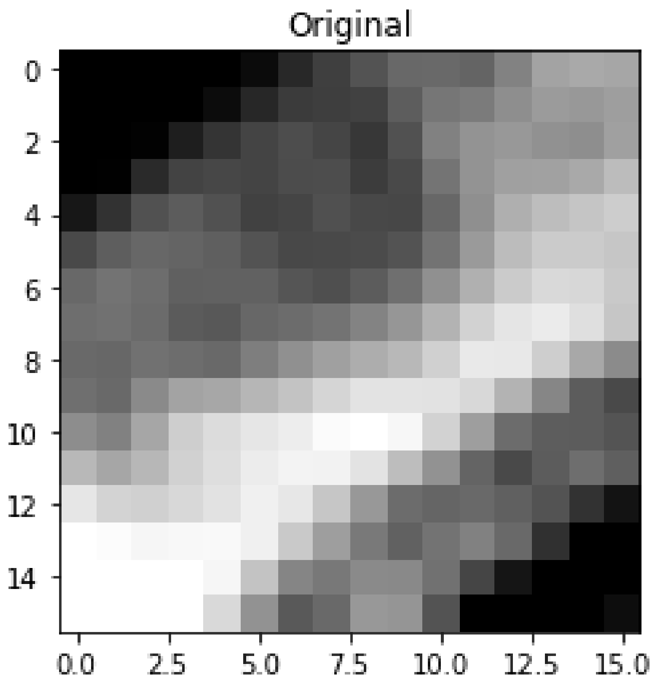
```
[14]: W1, B1, W2, B2 = We_t
      W2 = np.array(W2)
      W2 = W2.reshape(-1,16,16)

      figureNum += 1
      plt.figure(figureNum, figsize=(18, 16))

      for i in range(np.shape(W2)[0]):
          ax3 = plt.subplot(10, 8, i+1)
          ax3.imshow(W2[i], cmap='gray')
          ax3.set_yticks([])
          ax3.set_xticks([])
      plt.show()
```



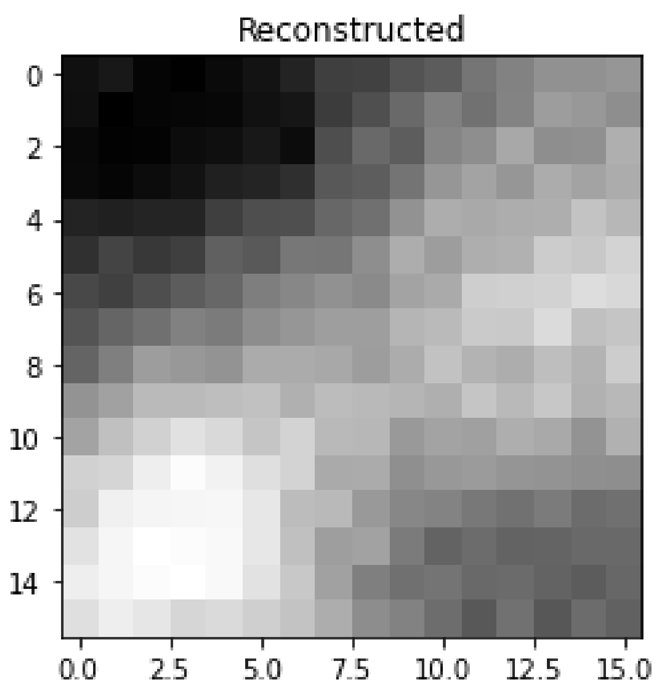
Here, we see that the weight matrices resemble the images. Also, it can be seen that there are certain patterns between some weights, which can mean that the given number of weights is enough or even maybe a little high for the network to learn the data. For the visualization of how the weights represent the data, a sample image is reconstructed below with the trained weights as:

```
[15]: sample_image = 92
      figureNum += 1
```

```
plt.figure(figureNum)
plt.imshow(data_final[sample_image], cmap='gray')
plt.title('Original')
plt.show(block=False)
```



[16]:
```
_,__,___, reconstructed_sample_image = forward(We_t, data_final_flat[sample_image])
figureNum += 1

reconstructed_sample_image = np.array(reconstructed_sample_image)
reconstructed_sample_image = reconstructed_sample_image.reshape(16,16)
plt.figure(figureNum)
plt.imshow(reconstructed_sample_image, cmap='gray')
plt.title('Reconstructed')
plt.show(block=False)
```



It can be seen that, since the reconsturected image does a great job of resembling the original image. However, there are some pixel intensity difference between the two of the images which is expected.

Part d

In this part, we are asked to retrain the network for 3 different values (low, medium, high) of $L_{hid} \in [10 \quad 100]$, of $\lambda \in [0 \quad 10^{-3}]$, while keeping β, ρ fixed. Then the hidden-layer features will be displayed as separate images. The hidden layer features are as shown below:

[17]:
```
Lin_l = Lpost_l = 16**2
Lhid_l = 12
lambdaa_l = 1e-2
beta_l = 0.001
rho_l = 0.2
params_l = [Lin_l, Lhid_l, lambdaa_l, beta_l, rho_l]
```

[18]:
```
loss_l, We_l = train_network(data_final_flat, params_l, 1e-2, 32, 50)
```

[19]:
```
Lin_m = Lpost_m = 16**2
Lhid_m = 50
lambdaa_m = 1e-2
beta_m = 0.001
rho_m = 0.2
params_m = [Lin_m, Lhid_m, lambdaa_m, beta_m, rho_m]
```

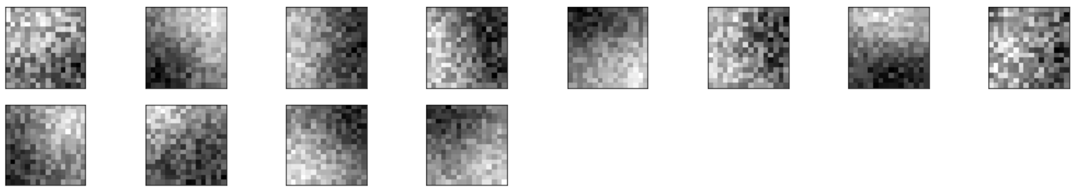
[20]:
```
loss_m, We_m = train_network(data_final_flat, params_m, 1e-2, 32, 50)
```

[21]:
```
Lin_h = Lpost_h = 16**2
Lhid_h = 98
lambdaa_h = 1e-2
beta_h = 0.001
rho_h = 0.2
```

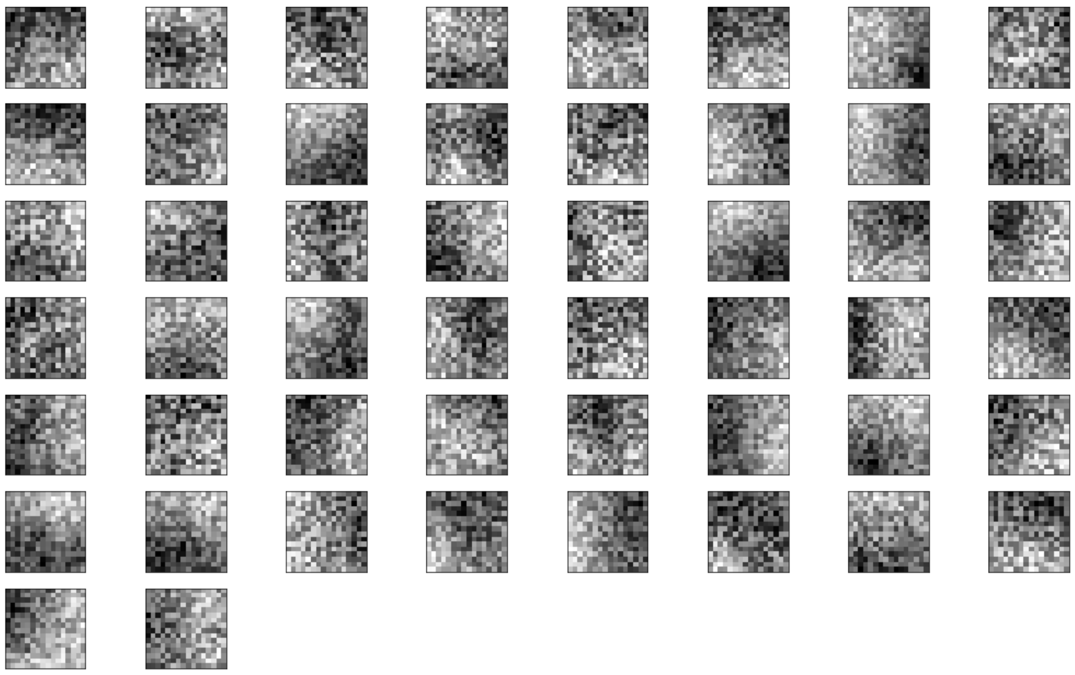
```
params_h = [Lin_h, Lhid_h, lambdaa_h, beta_h, rho_h]
```

[22]:
```
loss_h, We_h = train_network(data_final_flat, params_h, 1e-2, 32, 50)
```

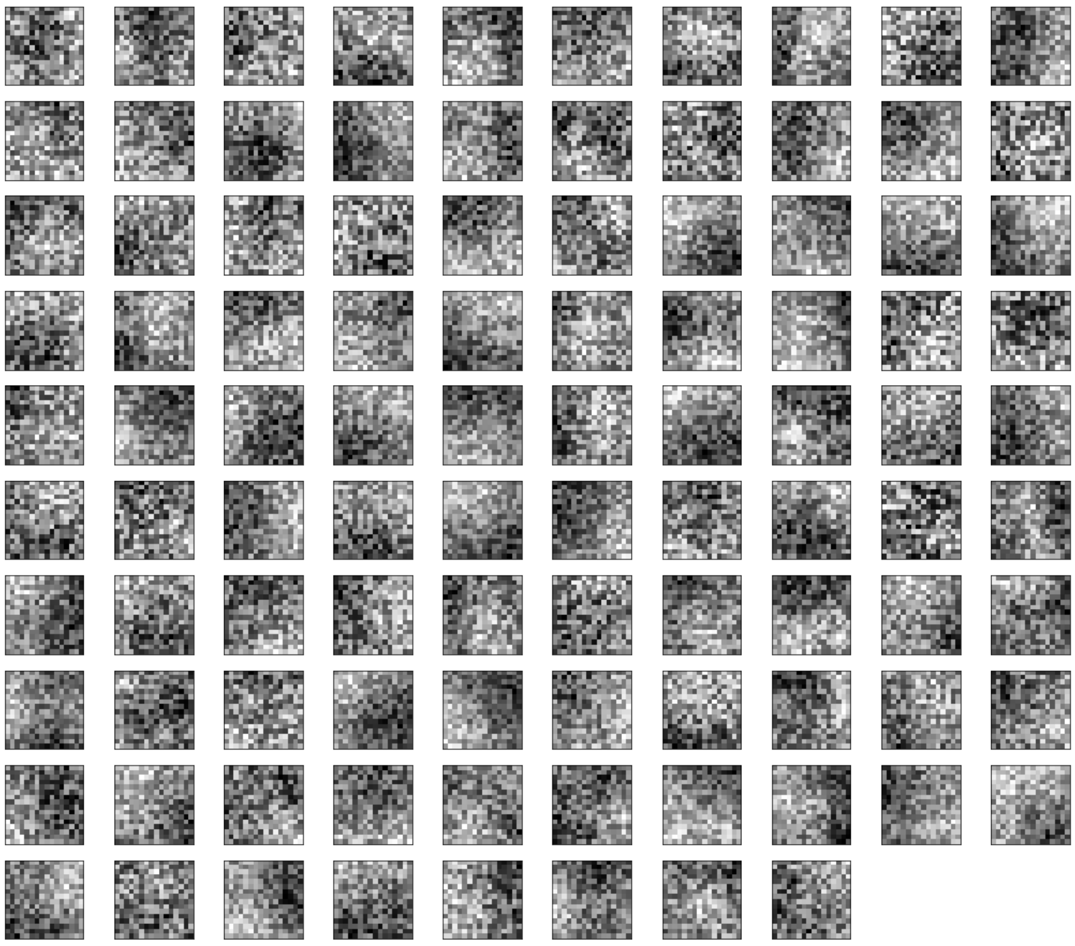
[23]:
```
W1_l, B1_l, W2_l, B2_l = We_l
W2_l = np.array(W2_l)
W2_l = W2_l.reshape(-1,16,16)

figureNum += 1
plt.figure(figureNum, figsize=(18, 16))

for i in range(np.shape(W2_l)[0]):
    ax3 = plt.subplot(10, 8, i+1)
    ax3.imshow(W2_l[i], cmap='gray')
    ax3.set_yticks([])
    ax3.set_xticks([])
plt.show()
```



[24]:
```
W1_m, B1_m, W2_m, B2_m = We_m
W2_m = np.array(W2_m)
W2_m = W2_m.reshape(-1,16,16)

figureNum += 1
plt.figure(figureNum, figsize=(18, 16))

for i in range(np.shape(W2_m)[0]):
    ax3 = plt.subplot(10, 8, i+1)
    ax3.imshow(W2_m[i], cmap='gray')
    ax3.set_yticks([])
    ax3.set_xticks([])
plt.show()
```



[25]:
```python
W1_h, B1_h, W2_h, B2_h = We_h
W2_h = np.array(W2_h)
W2_h = W2_h.reshape(-1,16,16)

figureNum += 1
plt.figure(figureNum, figsize=(18, 16))

for i in range(np.shape(W2_h)[0]):
    ax3 = plt.subplot(10, 10, i+1)
    ax3.imshow(W2_h[i], cmap='gray')
    ax3.set_yticks([])
    ax3.set_xticks([])
plt.show()
```



We can see from the 3 different model that the different number of hidden layer unit is affecting the overall learning process. As the number of neuron units increase, the model becomes more complex. However, when the number of neurons is increased too much, then the network is overfitting and that is clearly seen from the weights of the model with 98 hidden layer units. The network with 12 hidden layers is not enough to represent the images since the model is not complex enough.

Question 2

Part a

1 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.

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

[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,)
```

2 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:

[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))
```

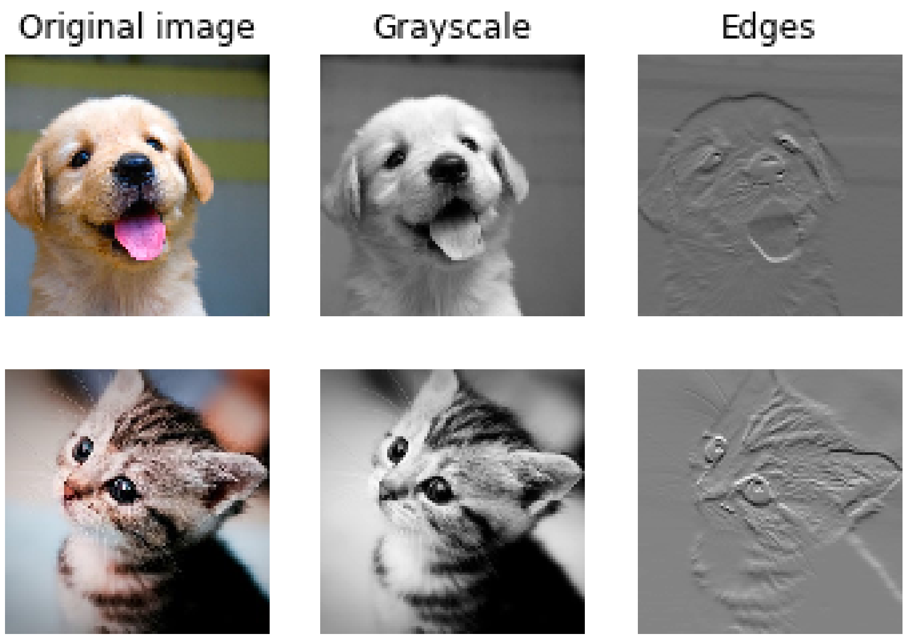
```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

3 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.

[4]:
```
from scipy.misc import imread, imresize

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, :, :, :] = imresize(puppy, (img_size, img_size)).transpose((2, 0, 1))
x[1, :, :, :] = imresize(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]]
```

```python
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()
```



4 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.

[5]:
```python
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)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], b, dout)

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
```

5 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:

[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
```

6 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:

[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], x, dout)

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
```

7 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:

[8]:
```python
# 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: 4.334410s
Fast: 0.007011s
Speedup: 618.235496x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 6.872578s
Fast: 0.008019s
Speedup: 857.065442x
dx difference:  1.949764775345631e-11
dw difference:  4.957046344783224e-13
db difference:  0.0
```

[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: 0.139665s
fast: 0.002955s
speedup: 47.268377x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.443848s
fast: 0.009974s
speedup: 44.502582x
dx difference:  0.0
```

8 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.

[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_param, pool_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b,␣
 ↪conv_param, pool_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b,␣
 ↪conv_param, pool_param)[0], b, dout)

# 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:  9.591132621921372e-09
dw error:  5.802401370096438e-09
db error:  1.0146343411762047e-09
```

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

# 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:  1.5218619980349303e-09
dw error:  2.702022646099404e-10
db error:  1.451272393591721e-10
```

9 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:

9.1 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.

[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
```

9.2 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.

[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=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
 ↪grads[param_name])))
```

```
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
```

9.3 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.

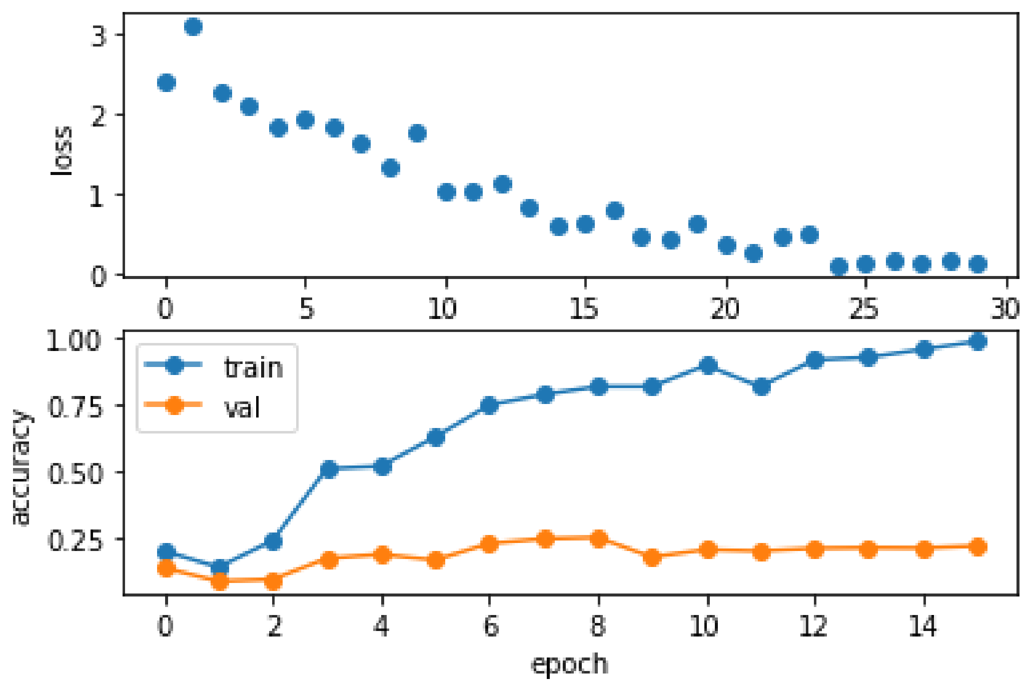
[14]:
```
np.random.seed(231)

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:

[15]:
```python
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()
```



9.4 Train the net

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

[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()
```

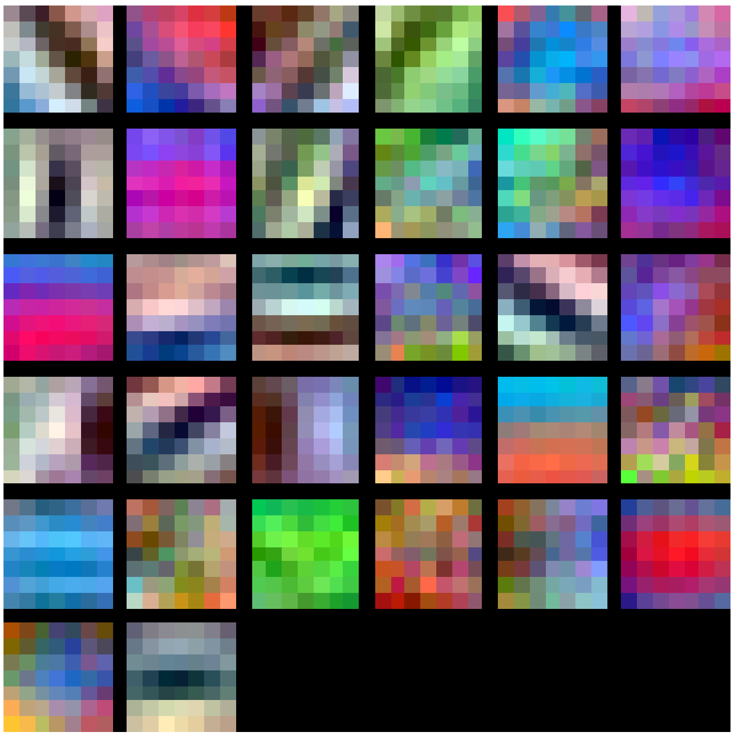
```
(Iteration 1 / 980) loss: 2.304740
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.098229
(Iteration 41 / 980) loss: 1.949788
(Iteration 61 / 980) loss: 1.888398
(Iteration 81 / 980) loss: 1.877093
(Iteration 101 / 980) loss: 1.851877
(Iteration 121 / 980) loss: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

9.5 Visualize Filters

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

[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()
```



10 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.

10.1 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:

[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]
```

[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]
```

10.2 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:

[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))
```

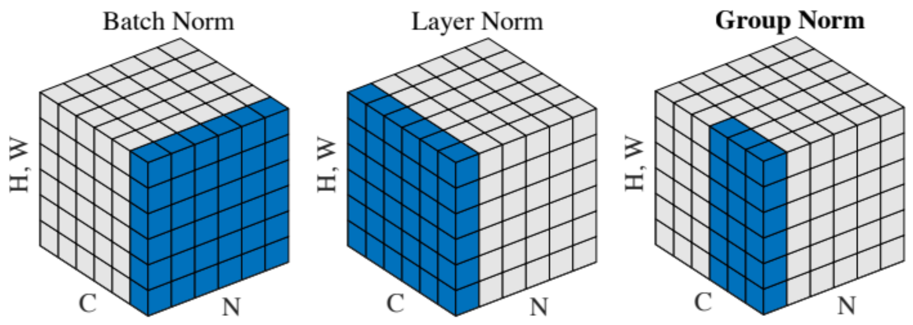
```
dx error:  3.083846820796372e-07
dgamma error:  7.09738489671469e-12
dbeta error:  3.275608725278405e-12
```

11 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.



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.

11.1 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:

[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]
```

11.2 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:

[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:  6.345904318452540e-08
dgamma error:  1.0546047434202244e-11
dbeta error:  3.810857316122484e-12
```

In this part, we are given a convolutional neural network to train on the CIFAR-10 dataset. First, edge detection and gray scaling is done. Then, a three layer conculutional neural network is trained with a data which is small in terms of size. After one epoch, we achieve 49.9% accuracy. Then the visualization of the first layer convolutional filters are shown. It can be seen from the filters that some of the filters are sensitive to the directional changes.

Then, spatial batch normalization is applied to data. The convolutional layers accept the inputs of shape (N, C, H, W) where the N is the minibatch size and the (H, W) is the spatial size of the feature map. The output is also shape of (N, C, H, W). Afterwards, the group normalization is introduced. In contrast to Layer Normalization, where the normalization is done over the entire feature per-datapoint, group normalization splits the each per-datapoint into G groups, and then does normalization over per-group per-datapoint instead.

Part b

12 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).

12.0.1 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.

12.0.2 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.

12.0.3 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.

12.1 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.

13 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

14 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.

[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
```

[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.

[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
```

15 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.

15.0.1 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).

[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]])
```

15.0.2 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.

[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])
```

15.0.3 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!

[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).

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

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

         # you must calculate the shape of the tensor after two conv layers, before the␣
     ↪fully-connected layer
         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])
```

15.0.4 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

[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))
```

[8]:
```
tensor([[-0.2991,  0.2627,  0.6589,  0.7674,  0.1105],
        [ 0.2789,  0.6538,  1.7469,  0.1362,  0.3428],
        [-0.6091,  1.2438, -0.1710, -1.1828,  0.0779]], device='cuda:0',
       requires_grad=True)
```

15.0.5 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.

[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))
```

15.0.6 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.

[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()
```

15.0.7 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.

[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 = 3.2225
Checking accuracy on the val set
Got 146 / 1000 correct (14.60%)

Iteration 100, loss = 1.9796
Checking accuracy on the val set
Got 352 / 1000 correct (35.20%)

Iteration 200, loss = 1.6270
Checking accuracy on the val set
Got 332 / 1000 correct (33.20%)

Iteration 300, loss = 1.9046
Checking accuracy on the val set
Got 384 / 1000 correct (38.40%)
```

```
Iteration 400, loss = 1.5942
Checking accuracy on the val set
Got 371 / 1000 correct (37.10%)

Iteration 500, loss = 1.6655
Checking accuracy on the val set
Got 426 / 1000 correct (42.60%)

Iteration 600, loss = 1.7015
Checking accuracy on the val set
Got 429 / 1000 correct (42.90%)

Iteration 700, loss = 1.3874
Checking accuracy on the val set
Got 481 / 1000 correct (48.10%)
```

15.0.8 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.

[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 = 3.3125
Checking accuracy on the val set
Got 125 / 1000 correct (12.50%)

Iteration 100, loss = 1.6043
Checking accuracy on the val set
Got 354 / 1000 correct (35.40%)

Iteration 200, loss = 1.6591
Checking accuracy on the val set
Got 407 / 1000 correct (40.70%)

Iteration 300, loss = 1.6062
Checking accuracy on the val set
Got 452 / 1000 correct (45.20%)

Iteration 400, loss = 1.6525
Checking accuracy on the val set
Got 463 / 1000 correct (46.30%)

Iteration 500, loss = 1.5696
Checking accuracy on the val set
Got 453 / 1000 correct (45.30%)

Iteration 600, loss = 1.2848
Checking accuracy on the val set
Got 457 / 1000 correct (45.70%)

Iteration 700, loss = 1.4486
Checking accuracy on the val set
Got 475 / 1000 correct (47.50%)
```

16 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.

16.0.1 Module API: Two-Layer Network

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

[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␣
 ↪dimension 50
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_TwoLayerFC()
```

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

16.0.2 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.

[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, 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])
```

16.0.3 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.

[15]:
```python
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))
```

16.0.4 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.

[16]:
```python
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()
```

16.0.5 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.

[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 = 3.5693
Checking accuracy on validation set
Got 143 / 1000 correct (14.30)

Iteration 100, loss = 2.3064
Checking accuracy on validation set
Got 300 / 1000 correct (30.00)

Iteration 200, loss = 2.5478
Checking accuracy on validation set
Got 388 / 1000 correct (38.80)

Iteration 300, loss = 1.8626
Checking accuracy on validation set
Got 390 / 1000 correct (39.00)

Iteration 400, loss = 1.4990
Checking accuracy on validation set
Got 421 / 1000 correct (42.10)

Iteration 500, loss = 1.8938
Checking accuracy on validation set
Got 387 / 1000 correct (38.70)

Iteration 600, loss = 1.9080
Checking accuracy on validation set
Got 437 / 1000 correct (43.70)
```

```
Iteration 700, loss = 1.6333
Checking accuracy on validation set
Got 417 / 1000 correct (41.70)
```

16.0.6 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.

[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 = 3.0509
Checking accuracy on validation set
Got 105 / 1000 correct (10.50)

Iteration 100, loss = 1.7127
Checking accuracy on validation set
Got 340 / 1000 correct (34.00)

Iteration 200, loss = 1.8109
Checking accuracy on validation set
Got 387 / 1000 correct (38.70)

Iteration 300, loss = 1.7886
Checking accuracy on validation set
Got 405 / 1000 correct (40.50)

Iteration 400, loss = 1.8073
Checking accuracy on validation set
Got 430 / 1000 correct (43.00)

Iteration 500, loss = 1.6066
Checking accuracy on validation set
Got 444 / 1000 correct (44.40)

Iteration 600, loss = 1.7604
Checking accuracy on validation set
```

```
Got 440 / 1000 correct (44.00)

Iteration 700, loss = 1.6320
Checking accuracy on validation set
Got 456 / 1000 correct (45.60)
```

17 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.

17.0.1 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.

[19]:
```python
# 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 = 2.3981
Checking accuracy on validation set
Got 112 / 1000 correct (11.20)

Iteration 100, loss = 1.8979
Checking accuracy on validation set
Got 409 / 1000 correct (40.90)

Iteration 200, loss = 2.0224
```

```
Checking accuracy on validation set
Got 388 / 1000 correct (38.80)

Iteration 300, loss = 2.2191
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Iteration 400, loss = 1.4410
Checking accuracy on validation set
Got 431 / 1000 correct (43.10)

Iteration 500, loss = 1.9541
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Iteration 600, loss = 1.7502
Checking accuracy on validation set
Got 444 / 1000 correct (44.40)

Iteration 700, loss = 1.6722
Checking accuracy on validation set
Got 437 / 1000 correct (43.70)
```

17.0.2 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.

[20]:
```python
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 = 2.3184
Checking accuracy on validation set
Got 114 / 1000 correct (11.40)

Iteration 100, loss = 1.7017
Checking accuracy on validation set
Got 458 / 1000 correct (45.80)

Iteration 200, loss = 1.4631
Checking accuracy on validation set
Got 478 / 1000 correct (47.80)

Iteration 300, loss = 1.5173
Checking accuracy on validation set
Got 512 / 1000 correct (51.20)

Iteration 400, loss = 1.2108
Checking accuracy on validation set
Got 529 / 1000 correct (52.90)

Iteration 500, loss = 1.3160
Checking accuracy on validation set
Got 569 / 1000 correct (56.90)

Iteration 600, loss = 1.2681
Checking accuracy on validation set
Got 558 / 1000 correct (55.80)

Iteration 700, loss = 1.1394
Checking accuracy on validation set
Got 555 / 1000 correct (55.50)
```

18 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

18.0.1 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.

18.0.2 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.

18.0.3 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

18.0.4 Have fun and happy training!

[21]:
```
################################################################################
# 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)
)

fc = nn.Linear(64*4*4, 10)

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

learning_rate = 1e-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 = 2.4820
Checking accuracy on validation set
Got 101 / 1000 correct (10.10)

Iteration 0, loss = 0.8536
Checking accuracy on validation set
Got 646 / 1000 correct (64.60)

Iteration 0, loss = 0.6532
Checking accuracy on validation set
Got 657 / 1000 correct (65.70)

Iteration 0, loss = 0.6673
Checking accuracy on validation set
Got 710 / 1000 correct (71.00)

Iteration 0, loss = 0.6098
Checking accuracy on validation set
Got 706 / 1000 correct (70.60)

Iteration 0, loss = 0.6354
Checking accuracy on validation set
Got 731 / 1000 correct (73.10)

Iteration 0, loss = 0.3882
Checking accuracy on validation set
Got 726 / 1000 correct (72.60)

Iteration 0, loss = 0.4312
Checking accuracy on validation set
Got 751 / 1000 correct (75.10)

Iteration 0, loss = 0.5236
Checking accuracy on validation set
Got 733 / 1000 correct (73.30)

Iteration 0, loss = 0.5197
Checking accuracy on validation set
Got 744 / 1000 correct (74.40)
```

18.1 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: Describe what you did

18.2 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.

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

```
Checking accuracy on test set
Got 7321 / 10000 correct (73.21)
```

The one advantange of architectures like PyTorch and Tensorflow is that the training is much faster since the code will run on GPUs instead of CPUs. Since the GPUs are faster doing matrix calculations, the overall training time on GPUs are much faster.

First, using PyTorch Tensor, image data is reshaped to be used in the fully connected neural network. We have discussed that an image is stored in a Tensor of shape (N, C, H, W) where N is the number of datapoints and (H, W) are the intermediate feature map in pixels. Then, a two-layer-fully-connected network is defined. This network performs a forward pass of a two-layer-fully-connected ReLU network on a batch of data. After that, the same procedure is done with a three-layer convolutional neural network. The weights are initialized using the Kaiming normal initialization. For training, stochastic gradient descent without momentum is used. Loss function is the cross entropy which is used from torch.functional.cross_entropy.

After 1 epoch, the two layer network's accuracy is 48.10%. And after 1 epoch, the three layer convulutional network's accuracy is 47.50%.

Then, the PyTorch Module API is introduced, which provides optimizers such as RMSProp, Adagrad and Adam. Now, instead of SGD from scratch, the PyTorch Module is used. Then the training over the two and three layer networks are done again. After 1 epoch, the two layer network's accuracy is 41.70%. And after 1 epoch, the three layer convulutional network's accuracy is 45.60%.

Then, PyTorch Sequential API is introduced which allows you to define arbitrary learnable layers and their connectivity. Again the training over two and three layer network is done. After 1 epoch, the two layer network's accuracy is 43.70%. And after 1 epoch, the three layer convulutional network's accuracy is 55.50%.

Finally, the PyTorch is experimented with the CIFAR-10 data set and the accuracy of 73% - 74% is acquired.

Question 3

In this question, we are asked to classify human activity (downstairs = 1, jogging = 2, sitting = 3, standing = 4, upstairs = 5, and walking = 6) from movement signals measured with three sensors simultaneously. The training and testing data is pre divided and given. The training data consists of 3000 samples and 150 time series and 3 sensor data. Thus, the shape of the training data is (3000, 150, 3). The test data has 600 samples and it has size of (600, 150, 3). We are asked to implement the fundamental recurrent neural network architectures, which are trained with back propagation through time.

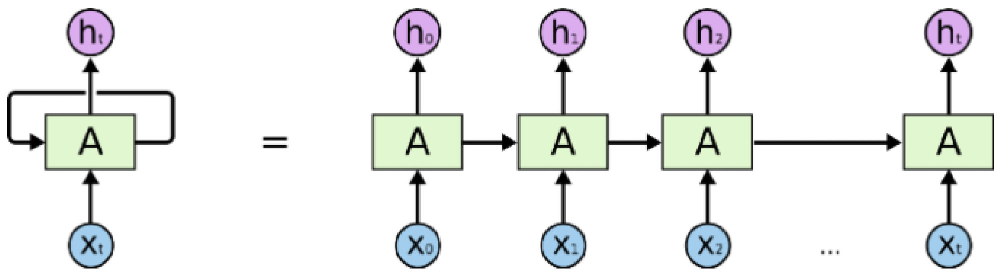
Part a

In this part, we are asked to implement a single layer recurrent neural network (RNN) with 128 neurons and tanh(hyperbolic tangent) as its activation function, followed by a multi layer percepton model with a softmax for classification. The training of the RNN will be done by implementing a back propagation through time algorithm. The stochastic gradient descent with mini-batch size of 32, learning rate of $\mu = 0.1$ and momentum rate of $\alpha = 0.85$ is used for training. The weights and biases will be initialized with Xavier Uniform distribution. The training will be done over maximum of 50 epochs and the algorithm will be stopped based on the categorical cross entropy over the validation data.

RNN structure

The difference between the RNN and neural network is that the, RNN can use their internal state to process sequences of inputs. This allows RNN to display dynamic temporal behaviour for a given time sequence. In other words, we can say that the RNN has memory and remembers everything that is happened before. Thus, the previous inputs effect the output of the current state since RNN remembers all the relations.

To have this kind of memory, RNN creates loops to preserve the past information.



This loop structure helps RNN to pass the information in one time step to another thus have memory.

The Python code for RNN is below [1][2]:

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

```
[2]: # Part A

     f = h5py.File('assign3_data3.h5', 'r')
     dataKeys = list(f.keys())
     print('The data keys are:' + str(dataKeys))

     # Gathering the  train images, test images, train labels and test labels.
     train_data = f['trX']
     train_labels = f['trY']
     test_data = f['tstX']
```

```
test_labels = f['tstY']

train_data = np.array(train_data)
train_labels = np.array(train_labels)
test_data = np.array(test_data)
test_labels = np.array(test_labels)

print('The size of train data is: ' + str(np.shape(train_data)))
print('The size of train labels is: ' + str(np.shape(train_labels)))
print('The size of test_data is: ' + str(np.shape(test_data)))
print('The size of test_labels is: ' + str(np.shape(test_labels)))
```

```
The data keys are:['trX', 'trY', 'tstX', 'tstY']
The size of train data is: (3000, 150, 3)
The size of train labels is: (3000, 6)
The size of test_data is: (600, 150, 3)
The size of test_labels is: (600, 6)
```

[3]:
```
def initialize_weights(fan_in, fan_out, wb_shape):

    np.random.seed(8)

    lim = np.sqrt(6/(fan_in+fan_out))

    weight = np.random.uniform(-lim, lim, size=(wb_shape))

    return weight
```

[6]:
```
class RNN:

    def __init__(self, input_dim = 3, hidden_dim = 128, seq_len = 150,␣
 ↪learning_rate = 1e-1,
                 momentumCoef = 0.85, output_class = 6, momentum_condition = False):

        np.random.seed(8)
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate

        self.momentumCoef = momentumCoef
        self.momentum_condition = momentum_condition
        self.last_t = 149

        # Weight initialization

        self.W1 = initialize_weights(self.input_dim, self.hidden_dim,  (self.
 ↪input_dim,self.hidden_dim))
        self.B1 = initialize_weights(self.input_dim, self.hidden_dim,  (1,self.
 ↪hidden_dim))

        self.W1_rec = initialize_weights(self.hidden_dim, self.hidden_dim,  (self.
 ↪hidden_dim,self.hidden_dim))
```

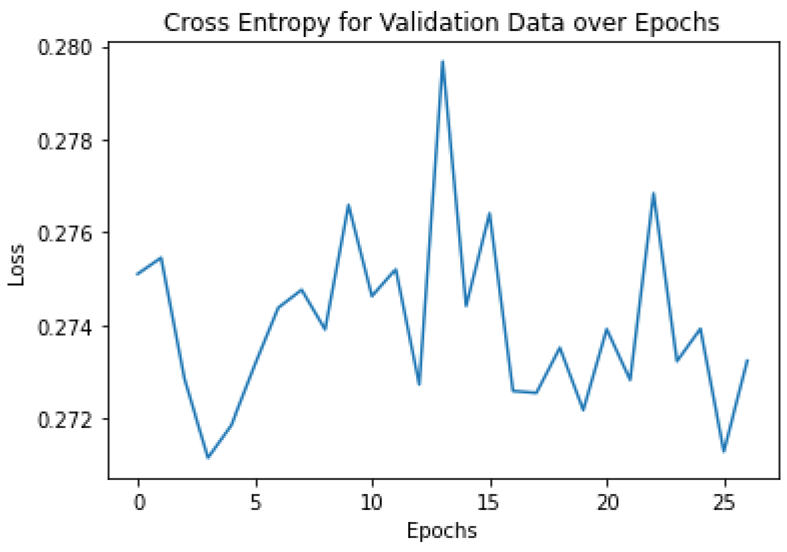
```
        self.W2 = initialize_weights(self.hidden_dim, self.output_class,  (self.
↪hidden_dim,self.output_class))
        self.B2 = initialize_weights(self.hidden_dim, self.output_class,  (1,self.
↪output_class))

        # momentum updates

        self.momentum_W1 = 0
        self.momentum_B1 = 0
        self.momentum_W1_rec = 0
        self.momentum_W2 = 0
        self.momentum_B2 = 0

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

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

        RETURNS:

                          : returns the accuracy between y and y_pred.
        '''
        count = 0
        for i in range(len(y)):
            if(y[i] == y_pred[i]):
                count += 1
        N = np.shape(y)[0]

        return 100 * (count / N)

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

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

        RETURNS:

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

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

    def sigmoid(self, x):

        '''
        This function is the sigmoid for the activation function.
        INPUTS:
            x            : x is the weighted sum which will be pushed to activation␣
↪function.

        RETURNS:

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

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

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

            x            : x is the input.

        RETURNS:

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


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

    def softmax(self, x):

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

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

        RETURNS:

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

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

    def der_softmax(self, x):

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

            x            : x is the input.

        RETURNS:

            result       : result is the derivative of softmax of the input x.
        '''
        p = self.softmax(x)
        result = p * (1-p)
        return result

    def CategoricalCrossEntropy(self, y, y_pred):

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

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

        RETURNS:

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

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

        cost = -np.mean(y * np.log(y_pred + 1e-15))
        return cost


    def forward(self, data):

        data_state = dict()
        hidden_state = dict()
        output_state = dict()
        probabilities = dict()

        self.h_prev_state = np.zeros((1,self.hidden_dim))
        hidden_state[-1] = self.h_prev_state
        # Loop over time T = 150 :

        for t in range(self.seq_len):

            data_state[t] = data[:,t]
            # Recurrent hidden layer computations:

            hidden_state[t] = self.tanh(np.dot(data_state[t], self.W1) + np.
↪dot(hidden_state[t-1], self.W1_rec) + self.B1)
            output_state[t] = np.dot(hidden_state[t], self.W2) + self.B2
            # The probabilities per class

            probabilities[t] = self.softmax(output_state[t])

        cache = [data_state, hidden_state, probabilities]
        return cache

    def BPTT(self,data,Y):

        cache = self.forward(data)

        data_state, hidden_state, probs = cache
        dW1, dW1_rec, dW2 = np.zeros((np.shape(self.W1))), np.zeros((np.shape(self.
↪W1_rec))), np.zeros((np.shape(self.W2)))
        dB1, dB2 = np.zeros((np.shape(self.B1))), np.zeros((np.shape(self.B2)))
        dhnext = np.zeros((np.shape(hidden_state[0])))

        dy = probs[self.last_t]
        dy[np.arange(len(Y)),np.argmax(Y,1)] -= 1
        dB2 += np.sum(dy,axis = 0, keepdims = True)


        dW2 += np.dot(hidden_state[self.last_t].T,dy)

        for t in reversed(range(1,self.seq_len)):
            dh = np.dot(dy, self.W2.T) + dhnext
            dh_rec = (1 - (hidden_state[t] * hidden_state[t])) * dh
            dB1 += np.sum(dh_rec,axis = 0, keepdims = True)
            dW1 += np.dot(data_state[t].T, dh_rec)
            dW1_rec += np.dot(hidden_state[t-1].T, dh_rec)
            dhnext = np.dot(dh_rec, self.W1_rec.T)

        grads = [dW1,dB1,dW1_rec,dW2,dB2]


        for grad in grads:
            np.clip(grad, -10, 10, out = grad)

        return grads, cache


    def update_weights(self,data,Y):

        grads, cache = self.BPTT(data,Y)
        dW1,dB1,dW1_rec,dW2,dB2 = grads
        sample_size = np.shape(cache)[0]
        # If momentum is used.
        if( self.momentum_condition == True ):

            self.momentum_W1 = dW1 + (self.momentumCoef * self.momentum_W1)
            self.momentum_B1 = dB1 + (self.momentumCoef * self.momentum_B1)
            self.momentum_W1_rec = dW1_rec + (self.momentumCoef * self.
↪momentum_W1_rec)
            self.momentum_W2 = dW2 + (self.momentumCoef * self.momentum_W2)
            self.momentum_B2 = dB2 + (self.momentumCoef * self.momentum_B2)

            self.W1 -= self.learning_rate * self.momentum_W1  /sample_size
            self.B1 -= self.learning_rate * self.momentum_B1 /sample_size
            self.W1_rec -= self.learning_rate * self.momentum_W1_rec /sample_size
            self.W2 -= self.learning_rate * self.momentum_W2 /sample_size
            self.B2 -= self.learning_rate * self.momentum_B2 /sample_size

        # If momentum is not used.
        else:

            self.W1 -= self.learning_rate * dW1 /sample_size
            self.B1 -= self.learning_rate * dB1 / sample_size
            self.W1_rec -= self.learning_rate * dW1_rec / sample_size
            self.W2 -= self.learning_rate * dW2 / sample_size
            self.B2 -= self.learning_rate * dB2 / sample_size

        return cache

    def train_network(self, data, labels, test_data, test_labels, epochs = 50,␣
↪batch_size = 32):

        np.random.seed(8)

        valid_loss = list()
        valid_accuracy = list()

        test_loss = list()
        test_accuracy = list()

        sample_size = np.shape(data)[0]
        k = int(sample_size / 10)

        for i in range(epochs):
            start_time = time.time()
            print('Epoch : ' +str(i))
            randomIndexes = np.random.permutation(sample_size)
            data = data[randomIndexes]

            number_of_batches = int(sample_size / batch_size)
            for j in range(number_of_batches):

                start = int(batch_size*j)
                end = int(batch_size*(j+1))

                data_feed = data[start:end]
                labels_feed = labels[start:end]

                cache_train = self.update_weights(data_feed, labels_feed)


            valid_data = data[0:k]
            valid_labels = labels[0:k]

            probs_valid, predictions_valid = self.predict(valid_data)

            cross_loss_valid = self.CategoricalCrossEntropy(valid_labels,␣
↪probs_valid[self.last_t])
            acc_valid = self.accuracy(np.argmax(valid_labels,1), predictions_valid)


            probs_test, predictions_test = self.predict(test_data)

            cross_loss_test = self.CategoricalCrossEntropy(test_labels,␣
↪probs_test[self.last_t])
            acc_test = self.accuracy(np.argmax(test_labels,1) ,predictions_test)

            valid_loss.append(cross_loss_valid)
            valid_accuracy.append(acc_valid)

            test_loss.append(cross_loss_test)
            test_accuracy.append(acc_test)

            end_time = time.time()
            print('Training time for 1 epoch : ' +str(end_time - start_time))
        valid_loss = np.array(valid_loss)
        valid_accuracy = np.array(valid_accuracy)

        test_loss = np.array(test_loss)
        test_accuracy = np.array(test_accuracy)

        return valid_loss, valid_accuracy, test_loss, test_accuracy

    def predict(self,X):

        cache = self.forward(X)
        probabilities = cache[-1]
        result = np.argmax(probabilities[self.last_t],axis=1)
        return probabilities, result
```

[7]:
```
RNN_model = RNN(input_dim = 3, hidden_dim = 128, learning_rate = 1e-12,␣
 ↪momentumCoef = 0.85,
                output_class = 6, momentum_condition = True)

valid_loss, valid_accuracy, test_loss, test_accuracy = RNN_model.
 ↪train_network(train_data, train_labels, test_data,                                                                                                                                                         ␣
 ↪test_labels, epochs = 27, batch_size = 32)
```

```
Epoch : 0
Training time for 1 epoch : 8.888329982757568
Epoch : 1
Training time for 1 epoch : 8.93514347076416
Epoch : 2
Training time for 1 epoch : 9.211557388305664
Epoch : 3
Training time for 1 epoch : 9.37718939781189
Epoch : 4
Training time for 1 epoch : 9.033752679824829
Epoch : 5
Training time for 1 epoch : 9.093088865280151
Epoch : 6
Training time for 1 epoch : 9.079053401947021
Epoch : 7
Training time for 1 epoch : 9.058162450790405
Epoch : 8
Training time for 1 epoch : 8.810735940933228
Epoch : 9
Training time for 1 epoch : 8.968312978744507
Epoch : 10
Training time for 1 epoch : 8.903622150421143
Epoch : 11
Training time for 1 epoch : 8.785381078720093
Epoch : 12
Training time for 1 epoch : 8.779760360717773
Epoch : 13
Training time for 1 epoch : 8.509472846984863
Epoch : 14
```

```
Training time for 1 epoch : 8.864723205566406
Epoch : 15
Training time for 1 epoch : 8.731741905212402
Epoch : 16
Training time for 1 epoch : 8.648432731628418
Epoch : 17
Training time for 1 epoch : 8.666646003723145
Epoch : 18
Training time for 1 epoch : 8.521836996078491
Epoch : 19
Training time for 1 epoch : 8.60711407661438
Epoch : 20
Training time for 1 epoch : 8.696110963821411
Epoch : 21
Training time for 1 epoch : 9.022264242172241
Epoch : 22
Training time for 1 epoch : 8.726747274398804
Epoch : 23
Training time for 1 epoch : 8.583361625671387
Epoch : 24
Training time for 1 epoch : 8.581809282302856
Epoch : 25
Training time for 1 epoch : 8.690657138824463
Epoch : 26
Training time for 1 epoch : 8.68241000175476
```

[67]:
```
figureNum = 0

plt.figure(figureNum)
plt.plot(valid_loss)

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Cross Entropy for Validation Data over Epochs')

plt.show()
```



[14]:
```
def confusion_matrix(labels, y_pred):
    labels_ = np.argmax(labels,1)
    result = np.zeros((6, 6))

    for i in range(len(labels_)):
        lab_i = labels_[i]
        y_pred_i = y_pred[i]
        result[lab_i,y_pred_i] +=1

    return result
```

[15]:
```
def accuracy_(confusion_matrix):
    accuracy = 0
    all_sum = 0
    for i in range(np.shape(confusion_matrix)[0]):
        for j in range(np.shape(confusion_matrix)[1]):
            all_sum += confusion_matrix[i,j]
            if (i == j):
                accuracy += confusion_matrix[i,j]

    return accuracy / all_sum * 100
```

[16]:
```
_,train_preds = RNN_model.predict(train_data)
_,test_preds = RNN_model.predict(test_data)

confusion_mat_train = confusion_matrix(train_labels,train_preds)

confusion_mat_test = confusion_matrix(test_labels,test_preds)
```

[17]:
```
accuracy_RNN_train = accuracy_(confusion_mat_train)
print('Accuracy of RNN with train data : ' +str(accuracy_RNN_train))
```

```
Accuracy of RNN with train data : 23.266666666666666
```

[18]:
```
accuracy_RNN_test = accuracy_(confusion_mat_test)
print('Accuracy of RNN with test data : ' +str(accuracy_RNN_test))
```

```
Accuracy of RNN with test data : 17.166666666666668
```

[21]:
```
print('Columns are : PREDICTION \n')
print('Rows are : ACTUAL \n')
print('The confusion matrix for the training data : \n \n'␣
 ↪+str(confusion_mat_train))
```

```
Columns are : PREDICTION

Rows are : ACTUAL

The confusion matrix for the training data :

[[143.  46. 161.  92.   3.  55.]
 [128. 118.  75.  94.  11.  74.]
 [  4. 151. 334.   2.   0.   9.]
 [ 15.  57. 403.  14.   8.   3.]
 [154.  76. 140.  74.   2.  54.]
 [177.  43.  82. 108.   3.  87.]]
```

```
[20]: print('Columns are : PREDICTION \n')
      print('Rows are : ACTUAL \n')

      print('The confusion matrix for the test data : \n \n' +str(confusion_mat_test))
```

```
The confusion matrix for the test data :

[[23.  8. 39. 26.  0.  4.]
 [34. 16.  9. 18.  2. 21.]
 [ 0. 53. 47.  0.  0.  0.]
 [37.  1. 60.  0.  0.  2.]
 [23.  8. 41. 19.  0.  9.]
 [35.  8. 16. 23.  1. 17.]]
```

The model is never get correct predictions for standing and upstairs in the test data and poorly predicts those classes in the training data. The accuracy for the training data is 23.26 and accuracy for the test data is 17.16. To compare my result, I have used the tensorflow's RNN structure.

Tensorflow Results

```
[52]: # Tensorflow

      import tensorflow as tf
      from tensorflow import keras
      from tensorflow.keras import layers
```

```
[54]: RNN = keras.Sequential()
      RNN.add(layers.SimpleRNN(128, batch_input_shape=[30, 150, 3]))
      RNN.add(layers.Dense(6, activation='softmax'))
      optimizer_RNN = keras.optimizers.SGD(learning_rate=0.1, momentum=0.85)
      RNN.compile(loss='categorical_crossentropy',
                            optimizer=optimizer_RNN,
                            metrics=['accuracy'])
```

```
[55]: model_RNN = RNN.fit(train_data, train_labels, batch_size=30, epochs=15)
```

```
Epoch 1/15
3000/3000 [==============================] - 5s 2ms/sample - loss: 3.1009 - acc:
0.2563
Epoch 2/15
3000/3000 [==============================] - 5s 2ms/sample - loss: 2.8241 - acc:
0.2490
Epoch 3/15
3000/3000 [==============================] - 4s 1ms/sample - loss: 2.7166 - acc:
0.2630
Epoch 4/15
3000/3000 [==============================] - 5s 2ms/sample - loss: 2.8876 - acc:
0.2523 1s - loss:
Epoch 5/15
3000/3000 [==============================] - 4s 1ms/sample - loss: 2.6660 - acc:
0.2637 1s - loss
Epoch 6/15
3000/3000 [==============================] - 5s 2ms/sample - loss: 2.8161 - acc:
0.2610 0s - loss: 2.7913 - acc: 0
Epoch 7/15
3000/3000 [==============================] - 5s 2ms/sample - loss: 2.8861 - acc:
0.2777 1s - loss: 2
Epoch 8/15
```

```
3000/3000 [==============================] - 5s 2ms/sample - loss: 2.4315 - acc:
0.2490
Epoch 9/15
3000/3000 [==============================] - 4s 1ms/sample - loss: 2.7921 - acc:
0.2683 1s - loss:
Epoch 10/15
3000/3000 [==============================] - 5s 2ms/sample - loss: 2.7909 - acc:
0.2967
Epoch 11/15
3000/3000 [==============================] - 5s 2ms/sample - loss: 2.3657 - acc:
0.3140
Epoch 12/15
3000/3000 [==============================] - 5s 2ms/sample - loss: 2.7846 - acc:
0.2953
Epoch 13/15
3000/3000 [==============================] - 5s 2ms/sample - loss: 2.3564 - acc:
0.3017 2s - loss: 2.3295 - acc: 0 - ETA: 2s - los
Epoch 14/15
3000/3000 [==============================] - 4s 1ms/sample - loss: 2.4632 - acc:
0.2850
Epoch 15/15
3000/3000 [==============================] - 5s 2ms/sample - loss: 2.3112 - acc:
0.2967
```

[56]:
```
train_preds_RNN = RNN.predict_classes(train_data, batch_size = 30)
test_preds_RNN = RNN.predict_classes(test_data, batch_size = 30)

confusion_mat_train_RNN = confusion_matrix(train_labels,train_preds_RNN)

confusion_mat_test_RNN = confusion_matrix(test_labels,test_preds_RNN)
```

[76]:
```
accuracy_RNN_train_tf = accuracy_(confusion_mat_train_RNN)
print('Accuracy of RNN with train data : ' +str(accuracy_RNN_train_tf))
```

```
Accuracy of RNN with train data : 35.6
```

[77]:
```
accuracy_RNN_test_tf = accuracy_(confusion_mat_test_RNN)
print('Accuracy of RNN with test data : ' +str(accuracy_RNN_test_tf))
```

```
Accuracy of RNN with test data : 29.833333333333336
```

[78]:
```
print('Columns are : PREDICTION \n')
print('Rows are : ACTUAL \n')

print('The confusion matrix(RNN) for the training data : \n \n'␣
 ↪+str(confusion_mat_train_RNN))
```

```
Columns are : PREDICTION

Rows are : ACTUAL

The confusion matrix(RNN) for the training data :

[[  1.  11.  56. 358.   0.  74.]
 [  2.  83.  59. 249.   0. 107.]
 [  0.  23. 386.  88.   0.   3.]
 [  0.   0.   9. 488.   0.   3.]
 [  0.  18.  37. 349.   0.  96.]
 [  0.  18.  29. 343.   0. 110.]]
```

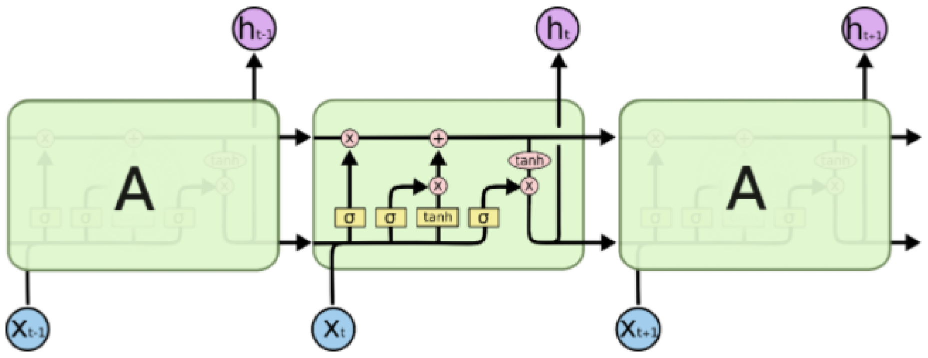
```
[79]: print('Columns are : PREDICTION \n')
      print('Rows are : ACTUAL \n')

      print('The confusion matrix(RNN) for the test data : \n \n'␣
       ↪+str(confusion_mat_test_RNN))
```

```
Columns are : PREDICTION

Rows are : ACTUAL

The confusion matrix(RNN) for the test data :

[[ 0.  2.  3. 88.  0.  7.]
 [ 0.  6. 17. 53.  0. 24.]
 [ 0.  0. 66. 34.  0.  0.]
 [ 0.  0.  1. 99.  0.  0.]
 [ 0.  0.  4. 90.  0.  6.]
 [ 0.  2. 10. 80.  0.  8.]]
```

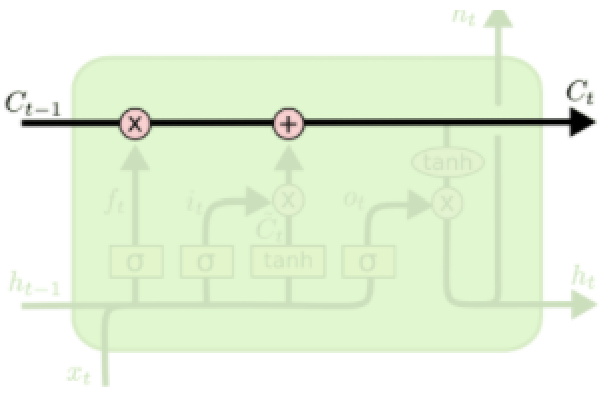
Accuracies for training and test data is better for the Tensorflow's RNN. However, the Tensorflow's RNN mostly predicts the third and fourth class which shows that the model is biased and did not learn so optimally.

Part b

In this part, we are asked to implement Long Short Term Memory networks(LSTM). The LSTM's are a kind of RNN, which are capable of learning long-term dependencies. In LSTM, instead of having a single neural network layer like in RNN, LSTM has 4 interactions in each module.



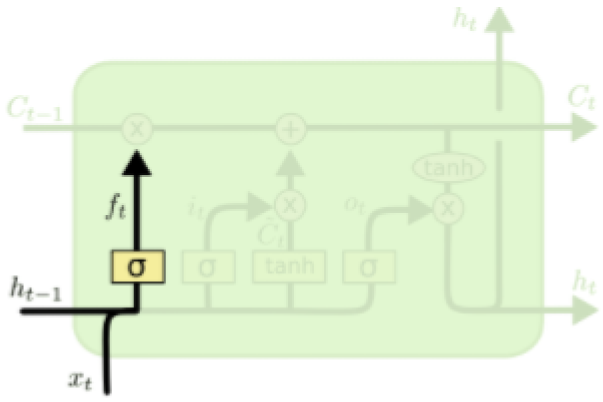
The horizontal line on the top of the figure that goes through each module in LSTM is the cell state. The main ideo of the cell state is that the information is carried along the modules with minor interactions. The cell state visualization can seen below:



The sigmoid layer that connects the cell state, clips the inputs that enter the cell state between the interval (0, 1). The '0' means does not let the information flow through the cell state and '1' means let everything pass through the cell state.

This decision of which information to get deleted and which information to keep is done through different gate layers.

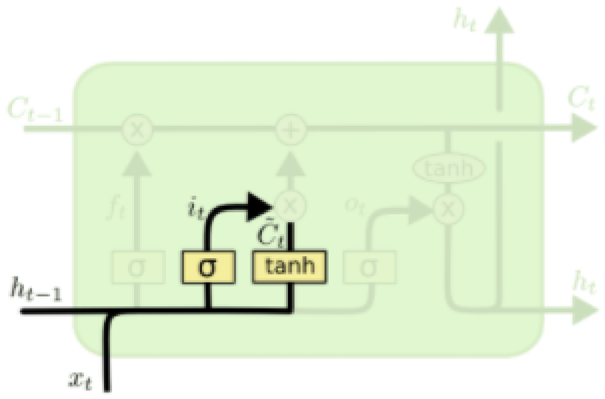
The *'forget gate layer'* is layer that decides which information to get deleted from the cell state. The *'forget gate layer'* uses sigmoid function to output from the interval (0, 1). As discussed before, '0' means does not let the information flow through the cell state and '1' means let everything pass through the cell state.



$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$$

where the f_t is the output of *'forget gate layer'*, W_f is the weights for *'forget gate layer'* and b_f is the bias for *'forget gate layer'*. This output is stored in the cell state C_{t-1}.

The *'input gate layer'* decides, which information to store in the cell state. the *'forget gate layer'* uses sigmoid function to decide which inputs to be update. The decision of which inputs to be selected is decided from the output of the sigmoid as discussed previously. Then a tanh layer creates a vector of new candidates $\hat{C}_t$



$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$$
$$\hat{C}_t = tanh(W_C.[h_{t-1}, x_t] + b_C)$$

where the i_t is the output of *'input gate layer'*, W_i is the weights for *'input gate layer'* and b_i is the bias for *'input gate layer'*. Also, C_t is the new cell state, W_C is the weights for the cell state and b_C is the bias for the cell state.

Then, the old cell state C_{t-1} will be updated using the function:

$$\hat{C}_t = f_t * C_{t-1} + i_t * \hat{C}_t$$

Finally, we will decide which parts of the cell state we are going to output. For that, a sigmoid function will be used for output and then we will put the cell state through tanh and multiply it with output so that only the decided parts are included. The function for that is as:

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * tanh(C_t)$$

The python implementation for that is [1][2][3]:

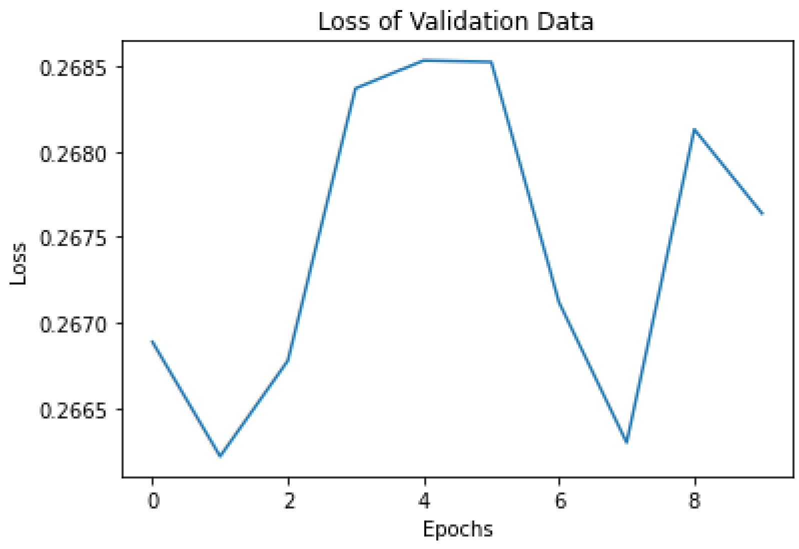
[22]:
```
class LSTM():

    def __init__(self,input_dim = 3,hidden_dim = 100, output_class = 6, seq_len =␣
↪150,
                 batch_size = 30, learning_rate = 1e-1, momentumCoef = 0.85,␣
↪momentum_condition = False):

        np.random.seed(150)

        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # Unfold case T = 150 :
        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.momentumCoef = momentumCoef
        self.momentum_condition = momentum_condition
        self.input_stack_dim = self.input_dim + self.hidden_dim
        self.last_t = 149
        # Weight initialization

        self.W_f = initialize_weights(self.input_dim, self.hidden_dim,  (self.
↪input_stack_dim,self.hidden_dim))
        self.B_f = initialize_weights(self.input_dim, self.hidden_dim,  (1,self.
↪hidden_dim))
        self.W_i = initialize_weights(self.input_dim, self.hidden_dim,  (self.
↪input_stack_dim,self.hidden_dim))
        self.B_i = initialize_weights(self.input_dim, self.hidden_dim,  (1,self.
↪hidden_dim))
        self.W_c = initialize_weights(self.input_dim, self.hidden_dim,  (self.
↪input_stack_dim,self.hidden_dim))
        self.B_c = initialize_weights(self.input_dim, self.hidden_dim,  (1,self.
↪hidden_dim))
        self.W_o = initialize_weights(self.input_dim, self.hidden_dim,  (self.
↪input_stack_dim,self.hidden_dim))
        self.B_o = initialize_weights(self.input_dim, self.hidden_dim,  (1,self.
↪hidden_dim))

        self.W = initialize_weights(self.hidden_dim, self.output_class,  (self.
↪hidden_dim, self.output_class))
        self.B = initialize_weights(self.hidden_dim, self.output_class,  (1, self.
↪output_class))


        # To keep previous updates in momentum :
```

```
        self.momentum_W_f = 0
        self.momentum_B_f = 0
        self.momentum_W_i = 0
        self.momentum_B_i = 0
        self.momentum_W_c = 0
        self.momentum_B_c = 0
        self.momentum_W_o = 0
        self.momentum_B_o = 0
        self.momentum_W = 0
        self.momentum_B = 0



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

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

        RETURNS:

                          : returns the accuracy between y and y_pred.
        '''
        count = 0
        for i in range(len(y)):
            if(y[i] == y_pred[i]):
                count += 1
        N = np.shape(y)[0]

        return 100 * (count / N)

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

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

        RETURNS:

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

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

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

        RETURNS:

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

    def sigmoid(self, x):

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

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

        RETURNS:

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

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

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

            x             : x is the input.

        RETURNS:

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


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

    def softmax(self, x):

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

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

        RETURNS:

            result        : result is the softmax of the input x.
        '''
        e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
        result = e_x / np.sum(e_x, axis=-1, keepdims=True)
        return result

    def der_softmax(self, x):

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

            x             : x is the input.

        RETURNS:

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

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

    def CategoricalCrossEntropy(self, y, y_pred):

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

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

        RETURNS:

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

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

        cost = -np.mean(y * np.log(y_pred + 1e-15))
        return cost



    def cell_forward(self,X,h_prev,C_prev):

        #print(X.shape,h_prev.shape)
        # Stacking previous hidden state vector with inputs:
        stack = np.column_stack([X, h_prev])

        # Forget gate:
        forget_gate = self.sigmoid(np.dot(stack,self.W_f) + self.B_f)

        # İnput gate:
        input_gate = self.sigmoid(np.dot(stack,self.W_i) + self.B_i)

        # New candidate:
        cell_bar = self.tanh(np.dot(stack,self.W_c) + self.B_c)
        # New Cell state:
        cell_state = forget_gate * C_prev + input_gate * cell_bar

        # Output fate:
        output_gate = self.sigmoid(np.dot(stack,self.W_o) + self.B_o)

        # Hidden state:
        hidden_state = output_gate * self.tanh(cell_state)

        # Classifiers (Softmax) :
        dense = np.dot(hidden_state, self.W) + self.B
        probs = self.softmax(dense)

        cache =␣
↪[stack,forget_gate,input_gate,cell_bar,cell_state,output_gate,hidden_state,dense,probs]
        return cache

    def forward(self, X, h_prev, C_prev):
        x_s,z_s,f_s,i_s = dict(), dict(), dict(), dict()
        C_bar_s,C_s, o_s, h_s = dict(), dict(), dict(), dict()
        v_s, y_s = dict(), dict()

        h_s[-1] = h_prev
        C_s[-1] = C_prev

        for t in range(150):

            x_s[t] = X[:,t,:]
            cache = self.cell_forward(x_s[t], h_s[t-1], C_s[t-1])
            z_s[t], f_s[t], i_s[t], C_bar_s[t], C_s[t], o_s[t],␣
↪h_s[t],v_s[t],y_s[t] = cache

        result_cache = [z_s, f_s, i_s, C_bar_s, C_s, o_s, h_s, v_s, y_s]
        return result_cache

    def BPTT(self, cache, Y):

        z_s, f_s, i_s, C_bar_s, C_s, o_s, h_s,v_s, y_s = cache

        dW_f = np.zeros((np.shape(self.W_f)))
        dW_i = np.zeros((np.shape(self.W_i)))
        dW_c = np.zeros((np.shape(self.W_c)))
        dW_o = np.zeros((np.shape(self.W_o)))
        dW = np.zeros((np.shape(self.W)))


        dB_f = np.zeros((np.shape(self.B_f)))
        dB_i = np.zeros((np.shape(self.B_i)))
        dB_c = np.zeros((np.shape(self.B_c)))
        dB_o = np.zeros((np.shape(self.B_o)))
        dB = np.zeros((np.shape(self.B)))

        dh_next = np.zeros(np.shape(h_s[0]))
        dC_next = np.zeros(np.shape(C_s[0]))

        # w.r.t. softmax input
        ddense = y_s[self.last_t]
        ddense[np.arange(len(Y)),np.argmax(Y,1)] -= 1

        # Softmax classifier's :

        dW = np.dot(h_s[149].T,ddense)
        dB = np.sum(ddense,axis = 0, keepdims = True)
        # Backprop through time:

        for t in reversed(range(1,150)):

            C_prev = C_s[t-1]

            # Output gate :
            dh = np.dot(ddense,self.W.T) + dh_next
            do = dh * self.tanh(C_s[t])
            do = do * self.der_sigmoid(o_s[t])
            dW_o += np.dot(z_s[t].T,do)
            dB_o += np.sum(do,axis = 0, keepdims = True)

            # Cell state:
            dC = dC_next
            dC += dh * o_s[t] * self.der_tanh(C_s[t])
            dC_bar = dC * i_s[t]
            dC_bar = dC_bar * self.der_tanh(C_bar_s[t])
            dW_c += np.dot(z_s[t].T,dC_bar)
            dB_c += np.sum(dC_bar,axis = 0, keepdims = True)

            # Input gate:
            di = dC * C_bar_s[t]
            di = self.der_sigmoid(i_s[t]) * di
            dW_i += np.dot(z_s[t].T,di)
            dB_i += np.sum(di,axis = 0,keepdims = True)

            # Forget gate:
            df = dC * C_prev
            df = df * self.der_sigmoid(f_s[t])
            dW_f += np.dot(z_s[t].T,df)
            dB_f += np.sum(df,axis = 0, keepdims = True)
            dz = np.dot(df,self.W_f.T) + np.dot(di,self.W_i.T) + np.dot(dC_bar,self.
↪W_c.T) + np.dot(do,self.W_o.T)
            dh_next = dz[:,-self.hidden_dim:]
            dC_next = f_s[t] * dC

        # List of gradients :
        grads = [dW,dB,dW_o,dB_o,dW_c,dB_c,dW_i,dB_i,dW_f,dB_f]

        # Clipping gradients anyway
        for grad in grads:
            np.clip(grad, -15, 15, out = grad)

        return h_s[self.last_t], C_s[self.last_t], grads

    def train_network(self, data, labels, test_data, test_labels, epochs = 50):

        valid_loss = list()
        valid_accuracy = list()
        test_loss = list()
        test_accuracy = list()

        sample_size = np.shape(data)[0]
        k = int(sample_size / 10)

        for epoch in range(epochs):

            print('Epoch : ' +str(epoch))

            randomIndexes = np.random.permutation(sample_size)
            data = data[randomIndexes]

            h_prev,C_prev = np.zeros((self.batch_size,self.hidden_dim)),np.
↪zeros((self.batch_size,self.hidden_dim))

            start_time = time.time()
            number_of_batches = int(sample_size / self.batch_size)
            for i in range(number_of_batches):

                start = int(self.batch_size*i)
                end = int(self.batch_size*(i+1))

                # Feeding random indexes:
                data_feed = data[start:end]
                labels_feed = labels[start:end]

                # Forward + BPTT + SGD:
                cache_train = self.forward(data_feed, h_prev, C_prev)
                h,c,grads = self.BPTT(cache_train, labels_feed)

                self.update_weights(grads)

                # Hidden state -------> Previous hidden state
                # Cell state ---------> Previous cell state
                h_prev,C_prev = h,c

            end_time = time.time()
            print('Training time for 1 epoch : ' +str(end_time - start_time))

            valid_data = data[0:k]
            valid_labels = labels[0:k]

            # Validation metrics calculations:

            valid_prevs = np.zeros((valid_data.shape[0],self.hidden_dim))

            valid_cache = self.forward(valid_data,valid_prevs,valid_prevs)
            probs_valid = valid_cache[-1]

            cross_loss_valid = self.CategoricalCrossEntropy(valid_labels,␣
↪probs_valid[self.last_t])


            # Test metrics calculations:
            test_prevs = np.zeros((test_data.shape[0],self.hidden_dim))
            test_cache = self.forward(test_data,test_prevs,test_prevs)
            probs_test = test_cache[-1]

            cross_loss_test = self.
↪CategoricalCrossEntropy(test_labels,probs_test[self.last_t])
            predictions_test = np.argmax(probs_test[self.last_t],1)
            acc_test = self.accuracy(np.argmax(test_labels,1),predictions_test)

            valid_loss.append(cross_loss_valid)
            test_loss.append(cross_loss_test)

            test_accuracy.append(acc_test)

        return valid_loss, test_loss, test_accuracy


    def update_weights(self, grads):

        dW,dB,dW_o,dB_o,dW_c,dB_c,dW_i,dB_i,dW_f,dB_f = grads

        # If momentum is used.
        if( self.momentum_condition == True ):

            self.momentum_W_f = dW_f + (self.momentumCoef * self.momentum_W_f)
            self.momentum_B_f = dB_f + (self.momentumCoef * self.momentum_B_f)
            self.momentum_W_i = dW_i + (self.momentumCoef * self.momentum_W_i)
            self.momentum_B_i = dB_i + (self.momentumCoef * self.momentum_B_i)
            self.momentum_W_c = dW_c + (self.momentumCoef * self.momentum_W_c)
            self.momentum_B_c = dB_c + (self.momentumCoef * self.momentum_B_c)
            self.momentum_W_o = dW_o + (self.momentumCoef * self.momentum_W_o)
            self.momentum_B_o = dB_o + (self.momentumCoef * self.momentum_B_o)
            self.momentum_W = dW + (self.momentumCoef * self.momentum_W)
            self.momentum_B = dB + (self.momentumCoef * self.momentum_B)


            self.W_f -= self.learning_rate * self.momentum_W_f
            self.B_f -= self.learning_rate * self.momentum_B_f
            self.W_i -= self.learning_rate * self.momentum_W_i
            self.B_i -= self.learning_rate * self.momentum_B_i
            self.W_c -= self.learning_rate * self.momentum_W_c
            self.B_c -= self.learning_rate * self.momentum_B_c
            self.W_o -= self.learning_rate * self.momentum_W_o
            self.B_o -= self.learning_rate * self.momentum_B_o
            self.W -= self.learning_rate * self.momentum_W
            self.B -= self.learning_rate * self.momentum_B

        # If momentum is not used.
        else:

            self.W_f -= self.learning_rate * dW_f
            self.B_f -= self.learning_rate * dB_f
            self.W_i -= self.learning_rate * dW_i
            self.B_i -= self.learning_rate * dB_i
            self.W_c -= self.learning_rate * dW_c
            self.B_c -= self.learning_rate * dB_c
            self.W_o -= self.learning_rate * dW_o
            self.B_o -= self.learning_rate * dB_o
            self.W -= self.learning_rate * dW
            self.B -= self.learning_rate * dB


    def predict(self,X):

        # Give zeros to hidden/cell states:
        pasts = np.zeros((np.shape(X)[0], self.hidden_dim))

        result_cache = self.forward(X,pasts,pasts)
        probabilities = result_cache[-1]
        result_prob = np.argmax(probabilities[self.last_t],axis=1)

        return result_prob
```

[37]:
```
LSTM_model = LSTM(learning_rate = 1e-15, momentumCoef = 0.85, batch_size = 32,␣
 ↪hidden_dim=128,  momentum_condition = True)

valid_loss_lstm, test_loss_lstm, test_accuracy_lstm = LSTM_model.
 ↪train_network(train_data, train_labels,                                      ␣

 ↪test_data, test_labels, epochs = 10)
```

```
Epoch : 0
Training time for 1 epoch : 57.761499881744385
Epoch : 1
Training time for 1 epoch : 76.37643480300903
Epoch : 2
Training time for 1 epoch : 92.29016256332397
Epoch : 3
Training time for 1 epoch : 73.1248984336853
Epoch : 4
Training time for 1 epoch : 91.22568488121033
Epoch : 5
Training time for 1 epoch : 74.34635710716248
Epoch : 6
Training time for 1 epoch : 91.12087869644165
Epoch : 7
Training time for 1 epoch : 78.14533972740173
Epoch : 8
Training time for 1 epoch : 84.42308497428894
Epoch : 9
Training time for 1 epoch : 77.23678541183472
```

[66]:
```
figureNum += 1

plt.figure(figureNum)
plt.plot(valid_loss_lstm)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss of Validation Data')
plt.show()
```



```
[41]: train_preds_lstm = LSTM_model.predict(train_data)
      test_preds_lstm = LSTM_model.predict(test_data)
      confusion_mat_train_lstm = confusion_matrix(train_labels, train_preds_lstm)
      confusion_mat_test_lstm = confusion_matrix(test_labels, test_preds_lstm)
```

```
[42]: accuracy_LSTM_train = accuracy_(confusion_mat_train_lstm)
      print('Accuracy of LSTM with train data : ' +str(accuracy_LSTM_train))
```

```
Accuracy of RNN with train data : 18.966666666666667
```

```
[43]: accuracy_LSTM_test = accuracy_(confusion_mat_test_lstm)
      print('Accuracy of LSTM with test data : ' +str(accuracy_LSTM_test))
```

```
Accuracy of RNN with test data : 21.333333333333336
```

```
[44]: print('Columns are : PREDICTION \n')
      print('Rows are : ACTUAL \n')

      print('The confusion matrix(LSTM) for the training data : \n \n'␣
        ↪+str(confusion_mat_train_lstm))
```

```
Columns are : PREDICTION

Rows are : ACTUAL

The confusion matrix(LSTM) for the training data :

[[121. 376.   3.   0.   0.   0.]
 [ 40. 448.   6.   6.   0.   0.]
 [405.  95.   0.   0.   0.   0.]
 [126. 372.   2.   0.   0.   0.]
 [ 99. 387.   9.   5.   0.   0.]
 [ 74. 414.  11.   1.   0.   0.]]
```

```
[45]: print('Columns are : PREDICTION \n')
      print('Rows are : ACTUAL \n')
print('The confusion matrix(LSTM) for the test data : \n \n'␣
↪+str(confusion_mat_test_lstm))
```

```
Columns are : PREDICTION

Rows are : ACTUAL

The confusion matrix(LSTM) for the test data :

[[39. 57.  4.  0.  0.  0.]
 [ 7. 89.  2.  2.  0.  0.]
 [78. 22.  0.  0.  0.  0.]
 [43. 57.  0.  0.  0.  0.]
 [26. 72.  2.  0.  0.  0.]
 [14. 85.  1.  0.  0.  0.]]
```

The accuracies for the train data is 18.96 and the accuracy for the test data is 21.33. It might seem that the LSTM performed better, however from the confusion matrix it is clearly seen that the LSTM predicts mostly 2 classes. This infers that the LSTM, we have implemented has not learned efficiently and it is biased.

To compare, I have implemented Tensorflow's LSTM and it is shown below:

Tensorflow Results

```
[59]: LSTM = keras.Sequential()
      LSTM.add(layers.LSTM(128, batch_input_shape=[30, 150, 3]))
      LSTM.add(layers.Dense(6, activation='softmax'))
      optimizer_LSTM = keras.optimizers.SGD(learning_rate=0.1, momentum=0.85)
      LSTM.compile(loss='categorical_crossentropy',
                              optimizer=optimizer_LSTM,
                              metrics=['accuracy'])
```

```
[60]: model_LSTM = LSTM.fit(train_data, train_labels, batch_size=30, epochs=15)
```

```
Epoch 1/15
3000/3000 [==============================] - 15s 5ms/sample - loss: 1.4074 -
acc: 0.4057
Epoch 2/15
3000/3000 [==============================] - 15s 5ms/sample - loss: 1.2501 -
acc: 0.4990
Epoch 3/15
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.0722 -
acc: 0.5693
Epoch 4/15
3000/3000 [==============================] - 15s 5ms/sample - loss: 1.0669 -
acc: 0.5667
Epoch 5/15
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.7537 -
acc: 0.3087
Epoch 6/15
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.5968 -
acc: 0.3770
Epoch 7/15
3000/3000 [==============================] - 15s 5ms/sample - loss: 1.5688 -
acc: 0.3923
Epoch 8/15
3000/3000 [==============================] - 15s 5ms/sample - loss: 1.3969 -
acc: 0.4363
```

```
Epoch 9/15
3000/3000 [==============================] - 15s 5ms/sample - loss: 1.3018 -
acc: 0.4870
Epoch 10/15
3000/3000 [==============================] - 15s 5ms/sample - loss: 1.2939 -
acc: 0.4807
Epoch 11/15
3000/3000 [==============================] - 15s 5ms/sample - loss: 1.1493 -
acc: 0.5380
Epoch 12/15
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.3209 -
acc: 0.5070
Epoch 13/15
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.1819 -
acc: 0.5383
Epoch 14/15
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.0963 -
acc: 0.5583
Epoch 15/15
3000/3000 [==============================] - 15s 5ms/sample - loss: 1.1229 -
acc: 0.5707
```

[61]:
```
train_preds_LSTM = LSTM.predict_classes(train_data, batch_size = 30)
test_preds_LSTM = LSTM.predict_classes(test_data, batch_size = 30)

confusion_mat_train_LSTM = confusion_matrix(train_labels,train_preds_LSTM)

confusion_mat_test_LSTM = confusion_matrix(test_labels,test_preds_LSTM)
```

[82]:
```
accuracy_LSTM_train_tf = accuracy_(confusion_mat_train_LSTM)
print('Accuracy of LSTM with train data : ' +str(accuracy_LSTM_train_tf))
```

```
Accuracy of LSTM with train data : 53.766666666666666
```

[83]:
```
accuracy_LSTM_test_tf = accuracy_(confusion_mat_test_LSTM)
print('Accuracy of LSTM with test data : ' +str(accuracy_LSTM_test_tf))
```

```
Accuracy of LSTM with test data : 47.833333333333336
```

[84]:
```
print('Columns are : PREDICTION \n')
print('Rows are : ACTUAL \n')

print('The confusion matrix(LSTM) for the training data : \n \n'␣
 ↪+str(confusion_mat_train_LSTM))
```

```
Columns are : PREDICTION

Rows are : ACTUAL

The confusion matrix(LSTM) for the training data :

[[261.  44.   6.  11.   0. 178.]
 [144. 299.   0.   0.   0.  57.]
 [  7.  10. 420.  21.   0.  42.]
 [ 30.   3.   3. 430.   0.  34.]
 [280.  63.   9.   9.   2. 137.]
 [242.  42.   8.   7.   0. 201.]]
```

[85]:
```
print('Columns are : PREDICTION \n')
print('Rows are : ACTUAL \n')

print('The confusion matrix(LSTM) for the test data : \n \n'␣
 ↪+str(confusion_mat_test_LSTM))
```

```
Columns are : PREDICTION

Rows are : ACTUAL

The confusion matrix(LSTM) for the test data :

[[67.  6.  2.  5.  0. 20.]
 [31. 52.  3.  0.  0. 14.]
 [ 2.  0. 68.  9.  0. 21.]
 [ 6.  0.  5. 73.  0. 16.]
 [62.  7.  0. 13.  0. 18.]
 [54. 11.  2.  6.  0. 27.]]
```

The accuracy for the train set is 53.76 and the accuracy for the test set is 47.83. Tensorflow's model is clearly working much better and it can be seen from the confusion matrices that the LSTM model of Tensorflow is predicting in a meaningful manner. However, it is not predicting the fifth class which is upstairs.

Part c

In this part, we are asked to implement an alternative to LSTM neural network called gated recurrent units (GRU).

The way GRU works is similar to LSTM, which lets the network to keep information from long time ago or remove irrelevant information like LSTM. However, GRU uses update gate and reset gate to achieve that.

Update gate z_t for time step t is calculated as:

$$z_t = \sigma(W_z x_t + U_z h_{t-1}])$$

h_{t-1} is the information for the t-1 unit and U_z is its own weights. Sigmoid is used for clipping the output into interval (0, 1). The update gate is used for determining how much of the information will be pass to future units.

Reset gate is similar to update gate, however the reset fate is used for deleting the data. The formula for the reset gate is:

$$r_t = \sigma(W_r x_t + U_r h_{t-1}])$$

which is similar to the update gate but this output will be used for forgetting the irrelevant information.

First, using reset gate a new memory content is introduced as:

$$\hat{h_t} = tanh(W x_t + r_t \odot U h_{t-1}])$$

The element-wise (Hadamard) product between the r_t and $U h_{t-1}$ determines what to information to remove from previous time steps.

Finally, the network calculates the h_t which is the vector that holds the information for the current unit. The formula for h_t is as:

$$h_t = z_t \odot h_{t_1} + (1 - z_t) \odot \hat{h_t}$$

For GRU, I could not implement the code from scratch, so I will only discuss Tensorflow's GRU network.

Tensorflow Results

```
[68]: GRU = keras.Sequential()
      GRU.add(layers.GRU(128, batch_input_shape=[30, 150, 3]))
      GRU.add(layers.Dense(6, activation='softmax'))
      optimizer_GRU = keras.optimizers.SGD(learning_rate=0.1, momentum=0.85)
      GRU.compile(loss='categorical_crossentropy',
                            optimizer=optimizer_GRU,
                            metrics=['accuracy'])

      model_GRU = GRU.fit(train_data, train_labels, batch_size=30, epochs=10)
```

```
Epoch 1/10
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.4369 -
acc: 0.4107
Epoch 2/10
3000/3000 [==============================] - 13s 4ms/sample - loss: 1.0227 -
acc: 0.5830
Epoch 3/10
3000/3000 [==============================] - 14s 5ms/sample - loss: 0.7887 -
acc: 0.6660
Epoch 4/10
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.4841 -
acc: 0.4937
Epoch 5/10
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.6470 -
acc: 0.4027
Epoch 6/10
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.8068 -
acc: 0.4087
Epoch 7/10
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.5963 -
acc: 0.4173
Epoch 8/10
3000/3000 [==============================] - 14s 5ms/sample - loss: 1.4922 -
acc: 0.4443
Epoch 9/10
3000/3000 [==============================] - 13s 4ms/sample - loss: 1.5514 -
acc: 0.4213
Epoch 10/10
3000/3000 [==============================] - 13s 4ms/sample - loss: 1.6748 -
acc: 0.4083
```

```
[69]: train_preds_GRU = GRU.predict_classes(train_data, batch_size=30)
      test_preds_GRU = GRU.predict_classes(test_data, batch_size=30)

      confusion_mat_train_GRU = confusion_matrix(train_labels,train_preds_GRU)

      confusion_mat_test_GRU = confusion_matrix(test_labels,test_preds_GRU)
```

```
[86]: accuracy_GRU_train_tf = accuracy_(confusion_mat_train_GRU)
      print('Accuracy of GRU with train data : ' +str(accuracy_GRU_train_tf))
```

```
Accuracy of GRU with train data : 35.53333333333333
```

```
[87]: accuracy_GRU_test_tf = accuracy_(confusion_mat_test_GRU)
      print('Accuracy of GRU with test data : ' +str(accuracy_GRU_test_tf))
```

```
Accuracy of GRU with test data : 25.833333333333336
```

```
[88]: print('Columns are : PREDICTION \n')
      print('Rows are : ACTUAL \n')

      print('The confusion matrix(GRU) for the training data : \n \n'␣
       ↪+str(confusion_mat_train_GRU))
```

```
Columns are : PREDICTION

Rows are : ACTUAL

The confusion matrix(GRU) for the training data :

[[290.  20.   0.   0.   2. 188.]
 [ 76. 255.   0.   0.  10. 159.]
 [  7.   1. 159.   0.   1. 332.]
 [ 97.   0.   2.   0.   0. 401.]
 [235.  45.   0.   0.  27. 193.]
 [147.  14.   0.   0.   4. 335.]]
```

```
[89]: print('Columns are : PREDICTION \n')
      print('Rows are : ACTUAL \n')

      print('The confusion matrix(GRU) for the test : \n \n' +str(confusion_mat_test_GRU))
```

```
Columns are : PREDICTION

Rows are : ACTUAL

The confusion matrix(GRU) for the test :

[[77.  1.  0.  0.  0. 22.]
 [20. 20.  0.  0.  9. 51.]
 [ 1.  0. 28.  0.  0. 71.]
 [ 2.  0.  0.  0.  0. 98.]
 [64.  0.  0.  0.  0. 36.]
 [69.  0.  0.  0.  1. 30.]]
```

The accuracy for the train data is 35.53 and the accuracy for the test data is 25.83. Also, the GRU does not predict the fourth and fifth class which are standing and upstairs.

Overall, the best performing method is the LSTM for both my implementation and tensorflow.

References

[1] *Victor Zhou*, "A Recurrent Neural Network (RNN) From Scratch" [Online] Available: https://github.com/vzhou842/rnn-from-scratch/blob/master/rnn.py

[2] “Building your Recurrent Neural Network - Step by Step¶,” Building a Recurrent Neural Network - Step by Step - v3. [Online]. Available: https://jmyao17.github.io/Machine_Learning/Sequence/RNN-1.html. [Accessed: 13-Dec-2020].

[3] *colah's blog*, "Understanding LSTM Networks" [Online] Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[4] *Simeon Kostadinov*, "Understanding GRU Networks" [Online] Available: https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be

Appendix

Question 1

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

```
[ ]: # Part A

     f = h5py.File('assign3_data1.h5', 'r')
     dataKeys = list(f.keys())
     print('The data keys are:' + str(dataKeys))

     # Gathering the  train images, test images, train labels and test labels.
     data = f['data']
     invXForm = f['invXForm']
     xForm = f['xForm']

     # data=np.array(data)
     # invXForm=np.array(invXForm)
     # xForm=np.array(xForm)

     # data = data.reshape(-1,16,16,3)

     print('The size of data is: ' + str(np.shape(data)))
     print('The size of invXForm is: ' + str(np.shape(invXForm)))
     print('The size of xForm is: ' + str(np.shape(xForm)))
```

```
[ ]: data_r = data[:,0,:,:]
     data_g = data[:,1,:,:]
     data_b = data[:,2,:,:]

     data_grayscale = data_r*0.2126 + data_g*0.7152 + data_b*0.0722
     print(np.shape(data_grayscale))
```

```
[ ]: def normalize_data(images):
         data_mean = np.mean(images, axis=(1,2))
         for i in range(np.shape(data_mean)[0]):
             images[i,:,:] -= data_mean[i]
         return images
```

```
[ ]: def map_std(images):
         data_std = np.std(images)
         mapped_data = np.where(images > 3*data_std, 3*data_std, images)
         mapped_data_final = np.where(mapped_data < -3*data_std, -3*data_std,␣
      ↪mapped_data)
         return mapped_data_final
```

```
[ ]: def clip_data_range(images, min_value, max_value):
         range_val = max_value - min_value
         max_data = np.max(images)
         min_data = np.min(images)

         result = images - min_data

         max_data = np.max(result)
```

```
    result = result / max_data * range_val

    result = result + min_value
    return result
```

[ ]:
```
data_grayscale_norm = normalize_data(data_grayscale)
data_grayscale_norm_mapped = map_std(data_grayscale_norm)
data_final = clip_data_range(data_grayscale_norm_mapped, 0.1, 0.9)
```

[ ]:
```
figureNum = 0
plt.figure(figureNum, figsize=(18, 16))
np.random.seed(9)
sample_size = np.shape(data_final)[0]
random_200 = np.random.randint(sample_size, size=(200))

for i,value in enumerate(random_200):
    ax1 = plt.subplot(20, 10,i+1)
    ax1.imshow(np.transpose(data[value], (1,2,0)))
    ax1.set_yticks([])
    ax1.set_xticks([])

plt.show()
```

[ ]:
```
figureNum += 1
plt.figure(figureNum, figsize=(18, 16))

for subplot,value in enumerate(random_200):
    ax2 = plt.subplot(20, 10,subplot+1)
    ax2.imshow(data_final[value], cmap='gray')
    ax2.set_yticks([])
    ax2.set_xticks([])
plt.show()
```

[ ]:
```
# Part B

def sigmoid(x):

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

def der_sigmoid(x):

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


def forward(We, data):

    W1, B1, W2, B2 = We

    # HIDDEN LAYER
    A1 = data.dot(W1) + B1
    Z1 = sigmoid(A1)
    # OUTPUT LAYER
    A2 = Z1.dot(W2) + B2
    y_pred = sigmoid(A2)
    return A1, Z1, A2, y_pred


def aeCost(We, data, params):
    Lin, Lhid, lambdaa, beta, rho = params
    W1, B1, W2, B2  = We
    sample_size = np.shape(data)[0]

    A1, Z1, A2, y_pred = forward(We, data)
    Z1_mean = np.mean(Z1, axis=0)

    J_1 = (1/(2*sample_size))*np.sum(np.power((data - y_pred),2))
    J_2 = (lambdaa / 2)* (np.sum(W1**2) + np.sum(W2**2))
    KL_1 = rho*np.log(Z1_mean/rho)
    KL_2 = (1-rho)*np.log((1-Z1_mean)/(1-rho))
    J_3 =  beta*np.sum(KL_1+KL_2)
    J = J_1 + J_2 - J_3

    del_out = -(data - y_pred) * der_sigmoid(y_pred)

    del_KL = beta*(-(rho/Z1_mean.T)+((1-rho)/(1-Z1_mean.T)))
    del_KLs =  np.vstack([del_KL]*sample_size)

    del_hidden = ((del_out.dot(W1)) + del_KLs) * der_sigmoid(Z1)

    # Gradients
    grad_W2 = (1/ sample_size)*(Z1.T.dot(del_out) + lambdaa*W2)
    grad_B2 = np.mean(del_out, axis=0, keepdims = True)

    grad_W1 = (1/ sample_size)*(data.T.dot(del_hidden) + lambdaa*W1)
    grad_B1 = np.mean(del_hidden, axis=0, keepdims = True)

    gradients = [grad_W2, grad_B2, grad_W1, grad_B1]

    return J, gradients


def update_weights(We, data, params, learning_rate):

    J, gradients = aeCost(We, data, params)
    grad_W2, grad_B2, grad_W1, grad_B1 = gradients
    W1, B1, W2, B2 = We

    # Update weights

    W2  -= learning_rate * grad_W2
    B2 -= learning_rate * grad_B2

    W1  -= learning_rate * grad_W1
    B1 -= learning_rate * grad_B1


    We_updated = [W1, B1, W2, B2]
    return J, We_updated
```

```
def initialize_weights(Lpre, Lhid):

    np.random.seed(8)

    Lpost = Lpre
    lim_1 = np.sqrt(6/(Lpre+Lhid))
    lim_2 = np.sqrt(6/(Lhid+Lpost))

    W1 = np.random.uniform(-lim_1, lim_1, (Lpre,Lhid))
    B1 = np.random.uniform(-lim_1, lim_1, (1,Lhid))

    W2 = np.random.uniform(-lim_2, lim_2, (Lhid,Lpost))
    B2 = np.random.uniform(-lim_2, lim_2, (1,Lpost))

    return W1, B1, W2, B2


def train_network(data, params, learning_rate, batch_size, epoch):

    np.random.seed(8)

    sample_size = np.shape(data)[0]
    Lin, Lhid, lambdaa, beta, rho = params
    W1, B1, W2, B2  = initialize_weights(Lin, Lhid)

    We = [W1, B1, W2, B2]
    Loss = list()
    for i in range(epoch):
        if(i % 10 == 0):
            print('Epoch: ' + str(i))
        # Randomize the dataset for each iteration
        randomIndexes = np.random.permutation(sample_size)
        data = data[randomIndexes]

        number_of_batches = int(sample_size / batch_size)

        for j in range(number_of_batches):

            # Mini batch start and end index
            start = int(batch_size*j)
            end = int(batch_size*(j+1))

            _, We = update_weights(We, data[start:end], params, learning_rate)

        J,_ = aeCost(We, data, params)
        Loss.append(J)

    return Loss, We
```

```
[ ]: data_final_flat = np.reshape(data_final, (np.shape(data_final)[0], 16**2))

Lin = Lpost = 16**2
Lhid = 64
lambdaa = 5e-4
beta = 0.01
rho = 0.2
params = [Lin, Lhid, lambdaa, beta, rho]
```

```
[ ]: loss, We_t = train_network(data_final_flat, params, 1e-2, 16, 80)
```

```
[ ]: figureNum += 1
     plt.figure(figureNum)
     plt.xlabel('Epoch')
     plt.ylabel('Loss')
     plt.title('Loss (aeLoss) over Epochs')
     plt.plot(loss)
     plt.show()
```

```
[ ]: W1, B1, W2, B2 = We_t
     W2 = np.array(W2)
     W2 = W2.reshape(-1,16,16)

     figureNum += 1
     plt.figure(figureNum, figsize=(18, 16))

     for i in range(np.shape(W2)[0]):
         ax3 = plt.subplot(10, 8, i+1)
         ax3.imshow(W2[i], cmap='gray')
         ax3.set_yticks([])
         ax3.set_xticks([])
     plt.show()
```

```
[ ]: sample_image = 92
     figureNum += 1
     plt.figure(figureNum)
     plt.imshow(data_final[sample_image], cmap='gray')
     plt.title('Original')
     plt.show(block=False)
```

```
[ ]: _,__,___, reconstructed_sample_image = forward(We_t, data_final_flat[sample_image])
     figureNum += 1

     reconstructed_sample_image = np.array(reconstructed_sample_image)
     reconstructed_sample_image = reconstructed_sample_image.reshape(16,16)
     plt.figure(figureNum)
     plt.imshow(reconstructed_sample_image, cmap='gray')
     plt.title('Reconstructed')
     plt.show(block=False)
```

```
[ ]: Lin_l = Lpost_l = 16**2
     Lhid_l = 12
     lambdaa_l = 1e-2
     beta_l = 0.001
     rho_l = 0.2
     params_l = [Lin_l, Lhid_l, lambdaa_l, beta_l, rho_l]
```

```
[ ]: loss_l, We_l = train_network(data_final_flat, params_l, 1e-2, 32, 50)
```

```
[ ]: Lin_m = Lpost_m = 16**2
     Lhid_m = 50
     lambdaa_m = 1e-2
     beta_m = 0.001
     rho_m = 0.2
     params_m = [Lin_m, Lhid_m, lambdaa_m, beta_m, rho_m]
```

[ ]:
```
loss_m, We_m = train_network(data_final_flat, params_m, 1e-2, 32, 50)
```

[ ]:
```
Lin_h = Lpost_h = 16**2
Lhid_h = 98
lambdaa_h = 1e-2
beta_h = 0.001
rho_h = 0.2
params_h = [Lin_h, Lhid_h, lambdaa_h, beta_h, rho_h]
```

[ ]:
```
loss_h, We_h = train_network(data_final_flat, params_h, 1e-2, 32, 50)
```

[ ]:
```
W1_l, B1_l, W2_l, B2_l = We_l
W2_l = np.array(W2_l)
W2_l = W2_l.reshape(-1,16,16)

figureNum += 1
plt.figure(figureNum, figsize=(18, 16))

for i in range(np.shape(W2_l)[0]):
    ax3 = plt.subplot(10, 8, i+1)
    ax3.imshow(W2_l[i], cmap='gray')
    ax3.set_yticks([])
    ax3.set_xticks([])
plt.show()
```

[ ]:
```
W1_m, B1_m, W2_m, B2_m = We_m
W2_m = np.array(W2_m)
W2_m = W2_m.reshape(-1,16,16)

figureNum += 1
plt.figure(figureNum, figsize=(18, 16))

for i in range(np.shape(W2_m)[0]):
    ax3 = plt.subplot(10, 8, i+1)
    ax3.imshow(W2_m[i], cmap='gray')
    ax3.set_yticks([])
    ax3.set_xticks([])
plt.show()
```

[ ]:
```
W1_h, B1_h, W2_h, B2_h = We_h
W2_h = np.array(W2_h)
W2_h = W2_h.reshape(-1,16,16)

figureNum += 1
plt.figure(figureNum, figsize=(18, 16))

for i in range(np.shape(W2_h)[0]):
    ax3 = plt.subplot(10, 10, i+1)
    ax3.imshow(W2_h[i], cmap='gray')
    ax3.set_yticks([])
    ax3.set_xticks([])
plt.show()
```

Question 3

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

[ ]:
```
# Part A

f = h5py.File('assign3_data3.h5', 'r')
dataKeys = list(f.keys())
print('The data keys are:' + str(dataKeys))

# Gathering the  train images, test images, train labels and test labels.
train_data = f['trX']
train_labels = f['trY']
test_data = f['tstX']
test_labels = f['tstY']

train_data = np.array(train_data)
train_labels = np.array(train_labels)
test_data = np.array(test_data)
test_labels = np.array(test_labels)

print('The size of train data is: ' + str(np.shape(train_data)))
print('The size of train labels is: ' + str(np.shape(train_labels)))
print('The size of test_data is: ' + str(np.shape(test_data)))
print('The size of test_labels is: ' + str(np.shape(test_labels)))
```

[ ]:
```
def initialize_weights(fan_in, fan_out, wb_shape):

    np.random.seed(8)

    lim = np.sqrt(6/(fan_in+fan_out))

    weight = np.random.uniform(-lim, lim, size=(wb_shape))

    return weight
```

[ ]:
```
class RNN:

    def __init__(self, input_dim = 3, hidden_dim = 128, seq_len = 150,␣
 ↪learning_rate = 1e-1,
                 momentumCoef = 0.85, output_class = 6, momentum_condition = False):

        np.random.seed(8)
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate

        self.momentumCoef = momentumCoef
        self.momentum_condition = momentum_condition
        self.last_t = 149
        # Weight initialization

        self.W1 = initialize_weights(self.input_dim, self.hidden_dim,  (self.
↪input_dim,self.hidden_dim))
        self.B1 = initialize_weights(self.input_dim, self.hidden_dim,  (1,self.
↪hidden_dim))

        self.W1_rec = initialize_weights(self.hidden_dim, self.hidden_dim,  (self.
↪hidden_dim,self.hidden_dim))

        self.W2 = initialize_weights(self.hidden_dim, self.output_class,  (self.
↪hidden_dim,self.output_class))
        self.B2 = initialize_weights(self.hidden_dim, self.output_class,  (1,self.
↪output_class))

        # momentum updates

        self.momentum_W1 = 0
        self.momentum_B1 = 0
        self.momentum_W1_rec = 0
        self.momentum_W2 = 0
        self.momentum_B2 = 0

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

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

        RETURNS:

                          : returns the accuracy between y and y_pred.
        '''
        count = 0
        for i in range(len(y)):
            if(y[i] == y_pred[i]):
                count += 1
        N = np.shape(y)[0]

        return 100 * (count / N)

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

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

        RETURNS:

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

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

    def sigmoid(self, x):

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

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

        RETURNS:

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

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

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

            x             : x is the input.

        RETURNS:

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


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

    def softmax(self, x):

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

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

        RETURNS:

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

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

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

            x             : x is the input.

        RETURNS:

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

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

    def CategoricalCrossEntropy(self, y, y_pred):

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

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

        RETURNS:

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

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

        cost = -np.mean(y * np.log(y_pred + 1e-15))
        return cost


    def forward(self, data):

        data_state = dict()
        hidden_state = dict()
        output_state = dict()
        probabilities = dict()

        self.h_prev_state = np.zeros((1,self.hidden_dim))
        hidden_state[-1] = self.h_prev_state
        # Loop over time T = 150 :

        for t in range(self.seq_len):

            data_state[t] = data[:,t]
            # Recurrent hidden layer computations:

            hidden_state[t] = self.tanh(np.dot(data_state[t], self.W1) + np.
↪dot(hidden_state[t-1], self.W1_rec) + self.B1)
            output_state[t] = np.dot(hidden_state[t], self.W2) + self.B2
            # The probabilities per class
            probabilities[t] = self.softmax(output_state[t])

        cache = [data_state, hidden_state, probabilities]
        return cache

    def BPTT(self,data,Y):

        cache = self.forward(data)

        data_state, hidden_state, probs = cache

        dW1, dW1_rec, dW2 = np.zeros((np.shape(self.W1))), np.zeros((np.shape(self.
↪W1_rec))), np.zeros((np.shape(self.W2)))
        dB1, dB2 = np.zeros((np.shape(self.B1))), np.zeros((np.shape(self.B2)))
        dhnext = np.zeros((np.shape(hidden_state[0])))

        dy = probs[self.last_t]
        dy[np.arange(len(Y)),np.argmax(Y,1)] -= 1
        dB2 += np.sum(dy,axis = 0, keepdims = True)


        dW2 += np.dot(hidden_state[self.last_t].T,dy)

        for t in reversed(range(1,self.seq_len)):
            dh = np.dot(dy, self.W2.T) + dhnext
            dh_rec = (1 - (hidden_state[t] * hidden_state[t])) * dh
            dB1 += np.sum(dh_rec,axis = 0, keepdims = True)
            dW1 += np.dot(data_state[t].T, dh_rec)
            dW1_rec += np.dot(hidden_state[t-1].T, dh_rec)
            dhnext = np.dot(dh_rec, self.W1_rec.T)

        grads = [dW1,dB1,dW1_rec,dW2,dB2]


        for grad in grads:
            np.clip(grad, -10, 10, out = grad)

        return grads, cache


    def update_weights(self,data,Y):

        grads, cache = self.BPTT(data,Y)
        dW1,dB1,dW1_rec,dW2,dB2 = grads
        sample_size = np.shape(cache)[0]
        # If momentum is used.
        if( self.momentum_condition == True ):

            self.momentum_W1 = dW1 + (self.momentumCoef * self.momentum_W1)
            self.momentum_B1 = dB1 + (self.momentumCoef * self.momentum_B1)
            self.momentum_W1_rec = dW1_rec + (self.momentumCoef * self.
↪momentum_W1_rec)
            self.momentum_W2 = dW2 + (self.momentumCoef * self.momentum_W2)
            self.momentum_B2 = dB2 + (self.momentumCoef * self.momentum_B2)

            self.W1 -= self.learning_rate * self.momentum_W1  /sample_size
            self.B1 -= self.learning_rate * self.momentum_B1 /sample_size
            self.W1_rec -= self.learning_rate * self.momentum_W1_rec /sample_size
            self.W2 -= self.learning_rate * self.momentum_W2 /sample_size
            self.B2 -= self.learning_rate * self.momentum_B2 /sample_size

        # If momentum is not used.
        else:

            self.W1 -= self.learning_rate * dW1 /sample_size
            self.B1 -= self.learning_rate * dB1 / sample_size
            self.W1_rec -= self.learning_rate * dW1_rec / sample_size
            self.W2 -= self.learning_rate * dW2 / sample_size
            self.B2 -= self.learning_rate * dB2 / sample_size

        return cache

    def train_network(self, data, labels, test_data, test_labels, epochs = 50,␣
↪batch_size = 32):

        np.random.seed(8)

        valid_loss = list()
        valid_accuracy = list()

        test_loss = list()
        test_accuracy = list()

        sample_size = np.shape(data)[0]
        k = int(sample_size / 10)

        for i in range(epochs):
            start_time = time.time()
            print('Epoch : ' +str(i))
            randomIndexes = np.random.permutation(sample_size)
            data = data[randomIndexes]

            number_of_batches = int(sample_size / batch_size)
            for j in range(number_of_batches):

                start = int(batch_size*j)
                end = int(batch_size*(j+1))

                data_feed = data[start:end]
                labels_feed = labels[start:end]

                cache_train = self.update_weights(data_feed, labels_feed)


            valid_data = data[0:k]
            valid_labels = labels[0:k]

            probs_valid, predictions_valid = self.predict(valid_data)

            cross_loss_valid = self.CategoricalCrossEntropy(valid_labels,␣
↪probs_valid[self.last_t])
            acc_valid = self.accuracy(np.argmax(valid_labels,1), predictions_valid)
            probs_test, predictions_test = self.predict(test_data)

            cross_loss_test = self.CategoricalCrossEntropy(test_labels,␣
↪probs_test[self.last_t])
            acc_test = self.accuracy(np.argmax(test_labels,1) ,predictions_test)

            valid_loss.append(cross_loss_valid)
            valid_accuracy.append(acc_valid)

            test_loss.append(cross_loss_test)
            test_accuracy.append(acc_test)

            end_time = time.time()
            print('Training time for 1 epoch : ' +str(end_time - start_time))
        valid_loss = np.array(valid_loss)
        valid_accuracy = np.array(valid_accuracy)

        test_loss = np.array(test_loss)
        test_accuracy = np.array(test_accuracy)

        return valid_loss, valid_accuracy, test_loss, test_accuracy

    def predict(self,X):

        cache = self.forward(X)
        probabilities = cache[-1]
        result = np.argmax(probabilities[self.last_t],axis=1)
        return probabilities, result
```

[ ]:
```
RNN_model = RNN(input_dim = 3, hidden_dim = 128, learning_rate = 1e-12,␣
↪momentumCoef = 0.85,
                output_class = 6, momentum_condition = True)

valid_loss, valid_accuracy, test_loss, test_accuracy = RNN_model.
↪train_network(train_data, train_labels, test_data,
                                                                              ␣
↪test_labels, epochs = 27, batch_size = 32)
```

[ ]:
```
figureNum = 0

plt.figure(figureNum)
plt.plot(valid_loss)

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Cross Entropy for Validation Data over Epochs')

plt.show()
```

[ ]:
```
def confusion_matrix(labels, y_pred):
    labels_ = np.argmax(labels,1)
    result = np.zeros((6, 6))

    for i in range(len(labels_)):
        lab_i = labels_[i]
        y_pred_i = y_pred[i]
        result[lab_i,y_pred_i] +=1

    return result
```

[ ]:
```
def accuracy_(confusion_matrix):
    accuracy = 0
    all_sum = 0
    for i in range(np.shape(confusion_matrix)[0]):
        for j in range(np.shape(confusion_matrix)[1]):
            all_sum += confusion_matrix[i,j]
            if (i == j):
                accuracy += confusion_matrix[i,j]

    return accuracy / all_sum * 100
```

[ ]:
```
_,train_preds = RNN_model.predict(train_data)
_,test_preds = RNN_model.predict(test_data)

confusion_mat_train = confusion_matrix(train_labels,train_preds)

confusion_mat_test = confusion_matrix(test_labels,test_preds)
```

[ ]:
```
accuracy_RNN_train = accuracy_(confusion_mat_train)
print('Accuracy of RNN with train data : ' +str(accuracy_RNN_train))
```

[ ]:
```
accuracy_RNN_test = accuracy_(confusion_mat_test)
print('Accuracy of RNN with test data : ' +str(accuracy_RNN_test))
```

[ ]:
```
print('Columns are : PREDICTION \n')
print('Rows are : ACTUAL \n')
print('The confusion matrix for the training data : \n \n'␣
 ↪+str(confusion_mat_train))
```

[ ]:
```
print('Columns are : PREDICTION \n')
print('Rows are : ACTUAL \n')

print('The confusion matrix for the test data : \n \n' +str(confusion_mat_test))
```

[ ]:
```
class LSTM():

    def __init__(self,input_dim = 3,hidden_dim = 100, output_class = 6, seq_len =␣
 ↪150,
                 batch_size = 30, learning_rate = 1e-1, momentumCoef = 0.85,␣
 ↪momentum_condition = False):

        np.random.seed(150)

        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # Unfold case T = 150 :
        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.momentumCoef = momentumCoef
        self.momentum_condition = momentum_condition
        self.input_stack_dim = self.input_dim + self.hidden_dim
        self.last_t = 149
        # Weight initialization

        self.W_f = initialize_weights(self.input_dim, self.hidden_dim,  (self.
↪input_stack_dim,self.hidden_dim))
        self.B_f = initialize_weights(self.input_dim, self.hidden_dim,  (1,self.
↪hidden_dim))
        self.W_i = initialize_weights(self.input_dim, self.hidden_dim,  (self.
↪input_stack_dim,self.hidden_dim))
        self.B_i = initialize_weights(self.input_dim, self.hidden_dim,  (1,self.
↪hidden_dim))
        self.W_c = initialize_weights(self.input_dim, self.hidden_dim,  (self.
↪input_stack_dim,self.hidden_dim))
        self.B_c = initialize_weights(self.input_dim, self.hidden_dim,  (1,self.
↪hidden_dim))
        self.W_o = initialize_weights(self.input_dim, self.hidden_dim,  (self.
↪input_stack_dim,self.hidden_dim))
        self.B_o = initialize_weights(self.input_dim, self.hidden_dim,  (1,self.
↪hidden_dim))

        self.W = initialize_weights(self.hidden_dim, self.output_class,  (self.
↪hidden_dim, self.output_class))
        self.B = initialize_weights(self.hidden_dim, self.output_class,  (1, self.
↪output_class))


        # To keep previous updates in momentum :
        self.momentum_W_f = 0
        self.momentum_B_f = 0
        self.momentum_W_i = 0
        self.momentum_B_i = 0
        self.momentum_W_c = 0
        self.momentum_B_c = 0
        self.momentum_W_o = 0
        self.momentum_B_o = 0
        self.momentum_W = 0
        self.momentum_B = 0



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

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

        RETURNS:

                        : returns the accuracy between y and y_pred.
        '''
        count = 0
        for i in range(len(y)):
            if(y[i] == y_pred[i]):
                count += 1
        N = np.shape(y)[0]

        return 100 * (count / N)

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

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

        RETURNS:

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

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

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

            x             : x is the input.

        RETURNS:

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

    def sigmoid(self, x):

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

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

        RETURNS:

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

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

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

            x             : x is the input.

        RETURNS:

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


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

    def softmax(self, x):

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

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

        RETURNS:

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

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

    def der_softmax(self, x):

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

            x             : x is the input.

        RETURNS:

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

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

    def CategoricalCrossEntropy(self, y, y_pred):

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

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

        RETURNS:

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

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

        cost = -np.mean(y * np.log(y_pred + 1e-15))
        return cost



    def cell_forward(self,X,h_prev,C_prev):

        #print(X.shape,h_prev.shape)
        # Stacking previous hidden state vector with inputs:
        stack = np.column_stack([X, h_prev])

        # Forget gate:
        forget_gate = self.sigmoid(np.dot(stack,self.W_f) + self.B_f)

        # İnput gate:
        input_gate = self.sigmoid(np.dot(stack,self.W_i) + self.B_i)

        # New candidate:
        cell_bar = self.tanh(np.dot(stack,self.W_c) + self.B_c)

        # New Cell state:
        cell_state = forget_gate * C_prev + input_gate * cell_bar

        # Output fate:
        output_gate = self.sigmoid(np.dot(stack,self.W_o) + self.B_o)

        # Hidden state:
        hidden_state = output_gate * self.tanh(cell_state)

        # Classifiers (Softmax) :
        dense = np.dot(hidden_state, self.W) + self.B
        probs = self.softmax(dense)

        cache =␣
↪[stack,forget_gate,input_gate,cell_bar,cell_state,output_gate,hidden_state,dense,probs]
        return cache

    def forward(self, X, h_prev, C_prev):
        x_s,z_s,f_s,i_s = dict(), dict(), dict(), dict()
        C_bar_s,C_s, o_s, h_s = dict(), dict(), dict(), dict()
        v_s, y_s = dict(), dict()

        h_s[-1] = h_prev
        C_s[-1] = C_prev
        for t in range(150):

            x_s[t] = X[:,t,:]
            cache = self.cell_forward(x_s[t], h_s[t-1], C_s[t-1])
            z_s[t], f_s[t], i_s[t], C_bar_s[t], C_s[t], o_s[t],␣
↪h_s[t],v_s[t],y_s[t] = cache

        result_cache = [z_s, f_s, i_s, C_bar_s, C_s, o_s, h_s, v_s, y_s]
        return result_cache

    def BPTT(self, cache, Y):

        z_s, f_s, i_s, C_bar_s, C_s, o_s, h_s,v_s, y_s = cache

        dW_f = np.zeros((np.shape(self.W_f)))
        dW_i = np.zeros((np.shape(self.W_i)))
        dW_c = np.zeros((np.shape(self.W_c)))
        dW_o = np.zeros((np.shape(self.W_o)))
        dW = np.zeros((np.shape(self.W)))


        dB_f = np.zeros((np.shape(self.B_f)))
        dB_i = np.zeros((np.shape(self.B_i)))
        dB_c = np.zeros((np.shape(self.B_c)))
        dB_o = np.zeros((np.shape(self.B_o)))
        dB = np.zeros((np.shape(self.B)))

        dh_next = np.zeros(np.shape(h_s[0]))
        dC_next = np.zeros(np.shape(C_s[0]))

        # w.r.t. softmax input
        ddense = y_s[self.last_t]
        ddense[np.arange(len(Y)),np.argmax(Y,1)] -= 1

        # Softmax classifier's :

        dW = np.dot(h_s[149].T,ddense)
        dB = np.sum(ddense,axis = 0, keepdims = True)
        # Backprop through time:

        for t in reversed(range(1,150)):

            C_prev = C_s[t-1]

            # Output gate :
            dh = np.dot(ddense,self.W.T) + dh_next
            do = dh * self.tanh(C_s[t])
            do = do * self.der_sigmoid(o_s[t])
            dW_o += np.dot(z_s[t].T,do)
            dB_o += np.sum(do,axis = 0, keepdims = True)

            # Cell state:
            dC = dC_next
            dC += dh * o_s[t] * self.der_tanh(C_s[t])
            dC_bar = dC * i_s[t]
            dC_bar = dC_bar * self.der_tanh(C_bar_s[t])
            dW_c += np.dot(z_s[t].T,dC_bar)
            dB_c += np.sum(dC_bar,axis = 0, keepdims = True)

            # Input gate:
            di = dC * C_bar_s[t]
            di = self.der_sigmoid(i_s[t]) * di
            dW_i += np.dot(z_s[t].T,di)
            dB_i += np.sum(di,axis = 0,keepdims = True)

            # Forget gate:
            df = dC * C_prev
            df = df * self.der_sigmoid(f_s[t])
            dW_f += np.dot(z_s[t].T,df)
            dB_f += np.sum(df,axis = 0, keepdims = True)
            dz = np.dot(df,self.W_f.T) + np.dot(di,self.W_i.T) + np.dot(dC_bar,self.
↪W_c.T) + np.dot(do,self.W_o.T)
            dh_next = dz[:,-self.hidden_dim:]
            dC_next = f_s[t] * dC

        # List of gradients :
        grads = [dW,dB,dW_o,dB_o,dW_c,dB_c,dW_i,dB_i,dW_f,dB_f]

        # Clipping gradients anyway
        for grad in grads:
            np.clip(grad, -15, 15, out = grad)

        return h_s[self.last_t], C_s[self.last_t], grads

    def train_network(self, data, labels, test_data, test_labels, epochs = 50):

        valid_loss = list()
        valid_accuracy = list()

        test_loss = list()
        test_accuracy = list()

        sample_size = np.shape(data)[0]
        k = int(sample_size / 10)

        for epoch in range(epochs):

            print('Epoch : ' +str(epoch))

            randomIndexes = np.random.permutation(sample_size)
            data = data[randomIndexes]

            h_prev,C_prev = np.zeros((self.batch_size,self.hidden_dim)),np.
↪zeros((self.batch_size,self.hidden_dim))

            start_time = time.time()
            number_of_batches = int(sample_size / self.batch_size)
            for i in range(number_of_batches):

                start = int(self.batch_size*i)
                end = int(self.batch_size*(i+1))

                # Feeding random indexes:
```

```
                data_feed = data[start:end]
                labels_feed = labels[start:end]

                # Forward + BPTT + SGD:
                cache_train = self.forward(data_feed, h_prev, C_prev)
                h,c,grads = self.BPTT(cache_train, labels_feed)

                self.update_weights(grads)

                # Hidden state -------> Previous hidden state
                # Cell state ---------> Previous cell state
                h_prev,C_prev = h,c

            end_time = time.time()
            print('Training time for 1 epoch : ' +str(end_time - start_time))

            valid_data = data[0:k]
            valid_labels = labels[0:k]

            # Validation metrics calculations:

            valid_prevs = np.zeros((valid_data.shape[0],self.hidden_dim))

            valid_cache = self.forward(valid_data,valid_prevs,valid_prevs)
            probs_valid = valid_cache[-1]

            cross_loss_valid = self.CategoricalCrossEntropy(valid_labels,␣
↪probs_valid[self.last_t])


            # Test metrics calculations:
            test_prevs = np.zeros((test_data.shape[0],self.hidden_dim))

            test_cache = self.forward(test_data,test_prevs,test_prevs)
            probs_test = test_cache[-1]

            cross_loss_test = self.
↪CategoricalCrossEntropy(test_labels,probs_test[self.last_t])
            predictions_test = np.argmax(probs_test[self.last_t],1)
            acc_test = self.accuracy(np.argmax(test_labels,1),predictions_test)

            valid_loss.append(cross_loss_valid)
            test_loss.append(cross_loss_test)

            test_accuracy.append(acc_test)

        return valid_loss, test_loss, test_accuracy


    def update_weights(self, grads):

        dW,dB,dW_o,dB_o,dW_c,dB_c,dW_i,dB_i,dW_f,dB_f = grads

        # If momentum is used.
        if( self.momentum_condition == True ):

            self.momentum_W_f = dW_f + (self.momentumCoef * self.momentum_W_f)
            self.momentum_B_f = dB_f + (self.momentumCoef * self.momentum_B_f)
            self.momentum_W_i = dW_i + (self.momentumCoef * self.momentum_W_i)
            self.momentum_B_i = dB_i + (self.momentumCoef * self.momentum_B_i)
            self.momentum_W_c = dW_c + (self.momentumCoef * self.momentum_W_c)
            self.momentum_B_c = dB_c + (self.momentumCoef * self.momentum_B_c)
            self.momentum_W_o = dW_o + (self.momentumCoef * self.momentum_W_o)
            self.momentum_B_o = dB_o + (self.momentumCoef * self.momentum_B_o)
            self.momentum_W = dW + (self.momentumCoef * self.momentum_W)
            self.momentum_B = dB + (self.momentumCoef * self.momentum_B)


            self.W_f -= self.learning_rate * self.momentum_W_f
            self.B_f -= self.learning_rate * self.momentum_B_f
            self.W_i -= self.learning_rate * self.momentum_W_i
            self.B_i -= self.learning_rate * self.momentum_B_i
            self.W_c -= self.learning_rate * self.momentum_W_c
            self.B_c -= self.learning_rate * self.momentum_B_c
            self.W_o -= self.learning_rate * self.momentum_W_o
            self.B_o -= self.learning_rate * self.momentum_B_o
            self.W -= self.learning_rate * self.momentum_W
            self.B -= self.learning_rate * self.momentum_B

        # If momentum is not used.
        else:

            self.W_f -= self.learning_rate * dW_f
            self.B_f -= self.learning_rate * dB_f
            self.W_i -= self.learning_rate * dW_i
            self.B_i -= self.learning_rate * dB_i
            self.W_c -= self.learning_rate * dW_c
            self.B_c -= self.learning_rate * dB_c
            self.W_o -= self.learning_rate * dW_o
            self.B_o -= self.learning_rate * dB_o
            self.W -= self.learning_rate * dW
            self.B -= self.learning_rate * dB


    def predict(self,X):

        # Give zeros to hidden/cell states:
        pasts = np.zeros((np.shape(X)[0], self.hidden_dim))

        result_cache = self.forward(X,pasts,pasts)
        probabilities = result_cache[-1]
        result_prob = np.argmax(probabilities[self.last_t],axis=1)

        return result_prob
```

```
[ ]: LSTM_model = LSTM(learning_rate = 1e-15, momentumCoef = 0.85, batch_size = 32,␣
      ↪hidden_dim=128,  momentum_condition = True)

     valid_loss_lstm, test_loss_lstm, test_accuracy_lstm = LSTM_model.
      ↪train_network(train_data, train_labels,
                                                                                    ␣
      ↪test_data, test_labels, epochs = 10)
```

```
[ ]: figureNum += 1

     plt.figure(figureNum)
     plt.plot(valid_loss_lstm)
     plt.xlabel('Epochs')
     plt.ylabel('Loss')
     plt.title('Loss of Validation Data')
     plt.show()
```

```
[ ]: train_preds_lstm = LSTM_model.predict(train_data)
     test_preds_lstm = LSTM_model.predict(test_data)
     confusion_mat_train_lstm = confusion_matrix(train_labels, train_preds_lstm)
     confusion_mat_test_lstm = confusion_matrix(test_labels, test_preds_lstm)
```

```
[ ]: accuracy_LSTM_train = accuracy_(confusion_mat_train_lstm)
     print('Accuracy of LSTM with train data : ' +str(accuracy_LSTM_train))
```

```
[ ]: accuracy_LSTM_test = accuracy_(confusion_mat_test_lstm)
     print('Accuracy of LSTM with test data : ' +str(accuracy_LSTM_test))
```

```
[ ]: print('Columns are : PREDICTION \n')
     print('Rows are : ACTUAL \n')

     print('The confusion matrix(LSTM) for the training data : \n \n'␣
      ↪+str(confusion_mat_train_lstm))
```

```
[ ]: print('Columns are : PREDICTION \n')
     print('Rows are : ACTUAL \n')

     print('The confusion matrix(LSTM) for the test data : \n \n'␣
      ↪+str(confusion_mat_test_lstm))
```

```
[ ]: # Tensorflow

     import tensorflow as tf
     from tensorflow import keras
     from tensorflow.keras import layers
```

```
[ ]: RNN = keras.Sequential()
     RNN.add(layers.SimpleRNN(128, batch_input_shape=[30, 150, 3]))
     RNN.add(layers.Dense(6, activation='softmax'))
     optimizer_RNN = keras.optimizers.SGD(learning_rate=0.1, momentum=0.85)
     RNN.compile(loss='categorical_crossentropy',
                       optimizer=optimizer_RNN,
                       metrics=['accuracy'])
```

```
[ ]: model_RNN = RNN.fit(train_data, train_labels, batch_size=30, epochs=15)
```

```
[ ]: train_preds_RNN = RNN.predict_classes(train_data, batch_size = 30)
     test_preds_RNN = RNN.predict_classes(test_data, batch_size = 30)

     confusion_mat_train_RNN = confusion_matrix(train_labels,train_preds_RNN)

     confusion_mat_test_RNN = confusion_matrix(test_labels,test_preds_RNN)
```

[ ]:
```
accuracy_RNN_train_tf = accuracy_(confusion_mat_train_RNN)
print('Accuracy of RNN with train data : ' +str(accuracy_RNN_train_tf))
```

[ ]:
```
accuracy_RNN_test_tf = accuracy_(confusion_mat_test_RNN)
print('Accuracy of RNN with test data : ' +str(accuracy_RNN_test_tf))
```

[ ]:
```
print('Columns are : PREDICTION \n')
print('Rows are : ACTUAL \n')

print('The confusion matrix(RNN) for the training data : \n \n'␣
 ↪+str(confusion_mat_train_RNN))
```

[ ]:
```
print('Columns are : PREDICTION \n')
print('Rows are : ACTUAL \n')

print('The confusion matrix(RNN) for the test data : \n \n'␣
 ↪+str(confusion_mat_test_RNN))
```

[ ]:
```
LSTM = keras.Sequential()
LSTM.add(layers.LSTM(128, batch_input_shape=[30, 150, 3]))
LSTM.add(layers.Dense(6, activation='softmax'))
optimizer_LSTM = keras.optimizers.SGD(learning_rate=0.1, momentum=0.85)
LSTM.compile(loss='categorical_crossentropy',
                      optimizer=optimizer_LSTM,
                      metrics=['accuracy'])
```

[ ]:
```
model_LSTM = LSTM.fit(train_data, train_labels, batch_size=30, epochs=15)
```

[ ]:
```
train_preds_LSTM = LSTM.predict_classes(train_data, batch_size = 30)
test_preds_LSTM = LSTM.predict_classes(test_data, batch_size = 30)

confusion_mat_train_LSTM = confusion_matrix(train_labels,train_preds_LSTM)

confusion_mat_test_LSTM = confusion_matrix(test_labels,test_preds_LSTM)
```

[ ]:
```
accuracy_LSTM_train_tf = accuracy_(confusion_mat_train_LSTM)
print('Accuracy of LSTM with train data : ' +str(accuracy_LSTM_train_tf))
```

[ ]:
```
accuracy_LSTM_test_tf = accuracy_(confusion_mat_test_LSTM)
print('Accuracy of LSTM with test data : ' +str(accuracy_LSTM_test_tf))
```

[ ]:
```
print('Columns are : PREDICTION \n')
print('Rows are : ACTUAL \n')

print('The confusion matrix(LSTM) for the training data : \n \n'␣
 ↪+str(confusion_mat_train_LSTM))
```

[ ]:
```
print('Columns are : PREDICTION \n')
print('Rows are : ACTUAL \n')

print('The confusion matrix(LSTM) for the test data : \n \n'␣
 ↪+str(confusion_mat_test_LSTM))
```

[ ]:
```
GRU = keras.Sequential()
GRU.add(layers.GRU(128, batch_input_shape=[30, 150, 3]))
GRU.add(layers.Dense(6, activation='softmax'))
```

```
optimizer_GRU = keras.optimizers.SGD(learning_rate=0.1, momentum=0.85)
GRU.compile(loss='categorical_crossentropy',
                      optimizer=optimizer_GRU,
                      metrics=['accuracy'])

model_GRU = GRU.fit(train_data, train_labels, batch_size=30, epochs=10)
```

```
[ ]: train_preds_GRU = GRU.predict_classes(train_data, batch_size=30)
     test_preds_GRU = GRU.predict_classes(test_data, batch_size=30)

     confusion_mat_train_GRU = confusion_matrix(train_labels,train_preds_GRU)

     confusion_mat_test_GRU = confusion_matrix(test_labels,test_preds_GRU)
```

```
[ ]: accuracy_GRU_train_tf = accuracy_(confusion_mat_train_GRU)
     print('Accuracy of GRU with train data : ' +str(accuracy_GRU_train_tf))
```

```
[ ]: accuracy_GRU_test_tf = accuracy_(confusion_mat_test_GRU)
     print('Accuracy of GRU with test data : ' +str(accuracy_GRU_test_tf))
```

```
[ ]: print('Columns are : PREDICTION \n')
     print('Rows are : ACTUAL \n')

     print('The confusion matrix(GRU) for the training data : \n \n'␣
      ↪+str(confusion_mat_train_GRU))
```

```
[ ]: print('Columns are : PREDICTION \n')
     print('Rows are : ACTUAL \n')

     print('The confusion matrix(GRU) for the test : \n \n' +str(confusion_mat_test_GRU))
```

```
[ ]:
```