EEE-443 Neural Networks Project-3

Name: Ayberk Yarkın

Surname: Yıldız

Id: 21803386

Date: December 28, 2021

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

Question 1

In this question, it is asked to implement an autoencoder neural network with a single hidden layer for the unsupervised feature extraction from natural images. The cost function to be minimized is given as:

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

I wrote the code to save my results in images that will save them in the same folder with the code script.

a) For part a, 200 random sample patches in RGB format are displayed. Additionally, the respective normalized versions of the same patches in grayscale format are displayed after evaluating the required steps. Results are shown below:

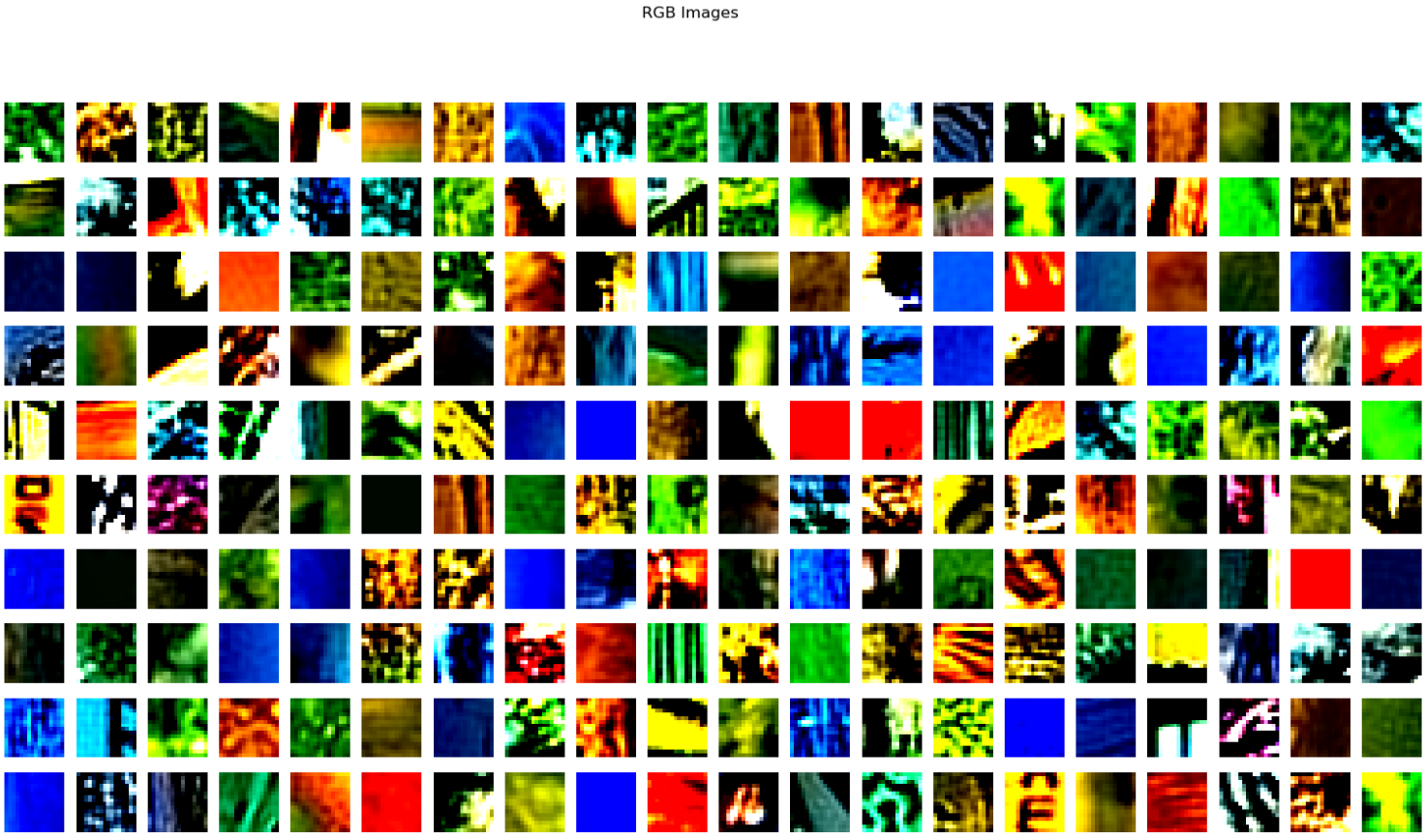


Figure 1: 200 random RGB images of the dataset

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

Figure 2: 200 random grayscaled images of the dataset

In the grayscale format, the shapes are the same with the ones in RGB format. Their only difference is that the grayscale formatted images are colorless. Images in grayscale format are easier to see and much more defined and apparent due to their colorless feature.

b) In this part, the required code is written for the network. The code has initialization according to the requirements given in the homework manual as:

$$w_o = \sqrt{\left(\frac{6}{L_{pre} + L_{post}}\right)} \quad (1)$$

where the interval is,

$$weights, bias = [-w_o, w_o] \quad (2)$$

The aeCost(W_e, data, params) function calculates the cost and its partial derivatives J and J_{grad} and returns them. Our weights are determined as:

$$W_e = [W_1\ W_2\ b_1\ b_2] \quad (3)$$

where the weights W_1, W_2 are transposes of each other due to the autoencoder fact. One of them is encoder and the other is the decoder.

Our data is at the size of $L_{in} \times N$ and $params$ include the parameters: $L_{in},\ L_{hid},\ lambda,\ beta,\ rho$. Training is done using the sigmoid function and J and J_{grad} are used in gradient descent solver to minimize the cost.

The average activation term $\hat{\rho}_b$ in the given cost equation is calculated as:

$$\hat{\rho}_b = \frac{1}{N}\sum_{n=0}^{N} h_n \quad (4)$$

The required hidden layer and lambda values with the optimized other parameters of the network and the cost function are:

Parameter	Value
$\boldsymbol{L_{in}}$	256
$\boldsymbol{L_{hid}}$	64
$\boldsymbol{\lambda}$	5×10^{-4}
$\boldsymbol{\beta}$	3
$\boldsymbol{\rho}$	0.03
$\boldsymbol{\alpha}$ **(Momentum Coefficient)**	0.8
$\boldsymbol{\eta}$ **(Learning Rate)**	0.1
Batch	32
Epoch	200

Table 1: Optimized Parameters for Question 1

Next part shows the results.

c) The first layer of connection weights for the neurons in the hidden layer is shown below:

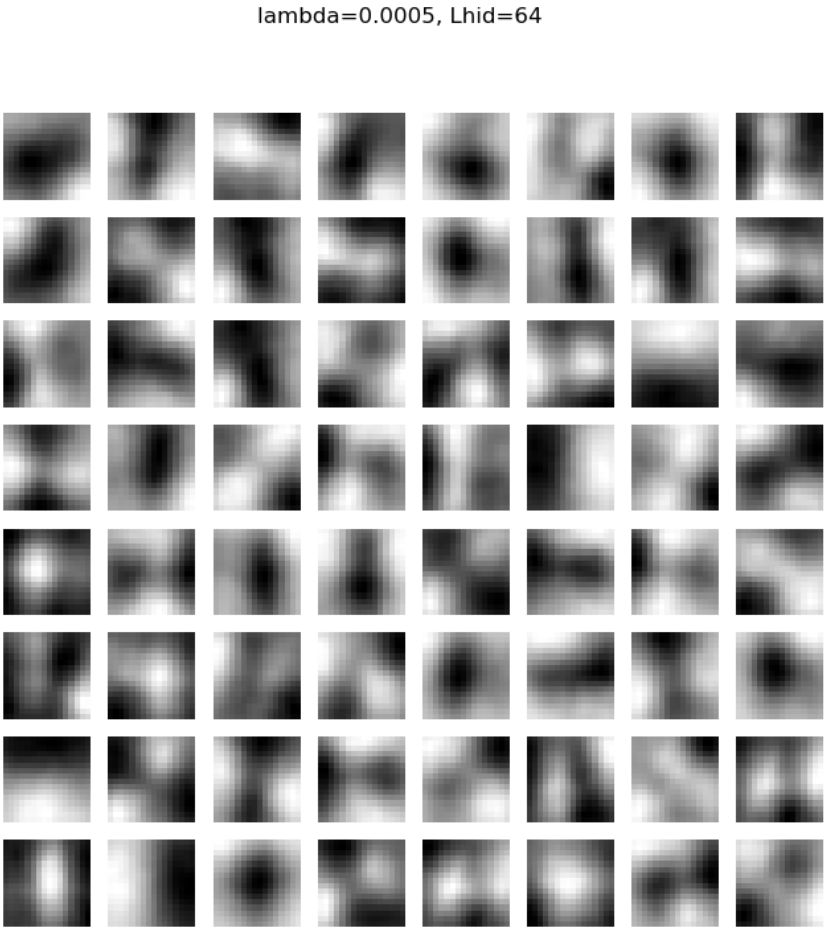


Figure 3: Images of first layer connection weights

Observing the result in Figure 3, there are variations in the images. There are some lines in different orientations, some curved lines and some different shaped parts. These images can combinely create a natural image, but they are not visualizing a natural image specifically. They are not representative of the natural images. The lines and curved lines can be combined and create a square, rectangle or a circle image in the dataset. Or the different shaped parts can be combined and create much complex shaped images in the dataset.

d) In this part, network is retrained with 3 different values of lambda and hidden layer units with determined low, medium and high values. β and ρ values are fixed. The intervals required for this purpose are: $L_{hid} \in [10\ 100]$ and $\lambda \in [0\ 10^{-3}]$. A total 9 combination of these values can be created. I determined my values as: $L_{hid} \in [16, 49, 100]$ and $\lambda \in [0, 10^{-5}, 10^{-3}]$. The hidden layer feature results can be seen below:

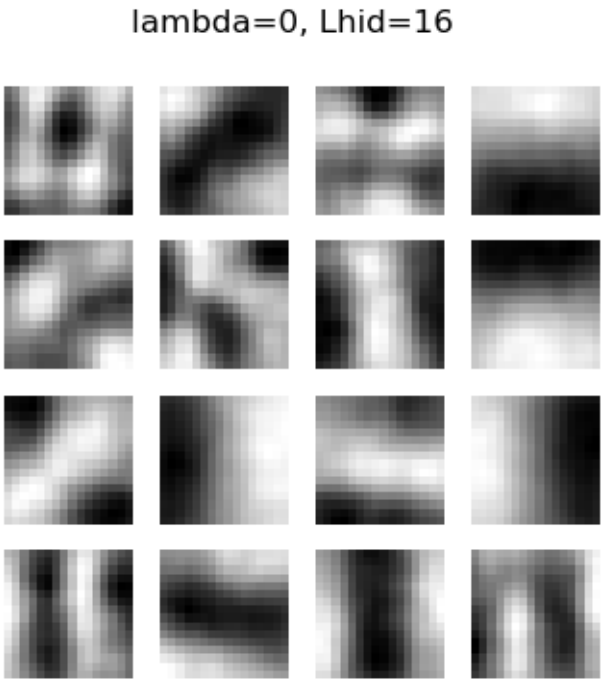


Figure 4: $\lambda = 0, L_{hid} = 16$

Figure 5: $\lambda = 0, L_{hid} = 49$

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

Figure 6: $\lambda = 0, L_{hid} = 100$

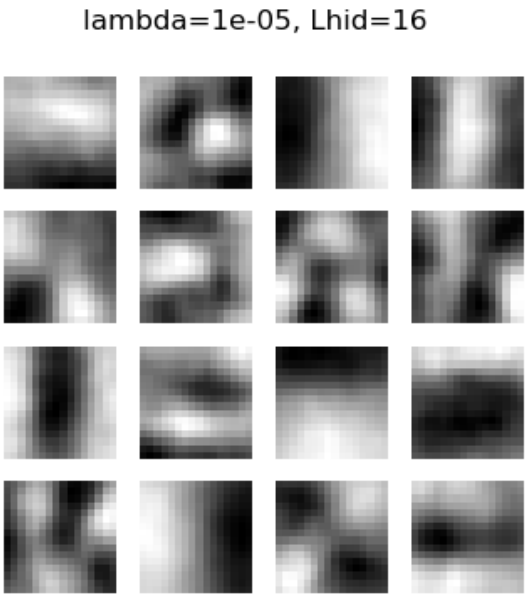


Figure 7: $\lambda = 10^{-5}, L_{hid} = 16$

Figure 8: $\lambda = 10^{-5}, L_{hid} = 49$

Figure 9: $\lambda = 10^{-5}, L_{hid} = 100$

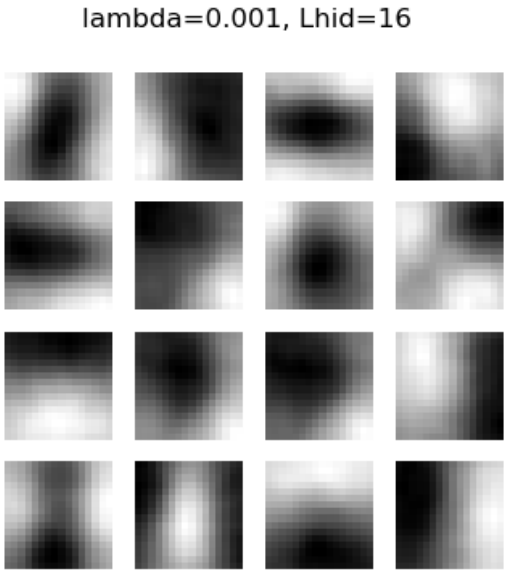


Figure 10: $\lambda = 10^{-3}, L_{hid} = 16$

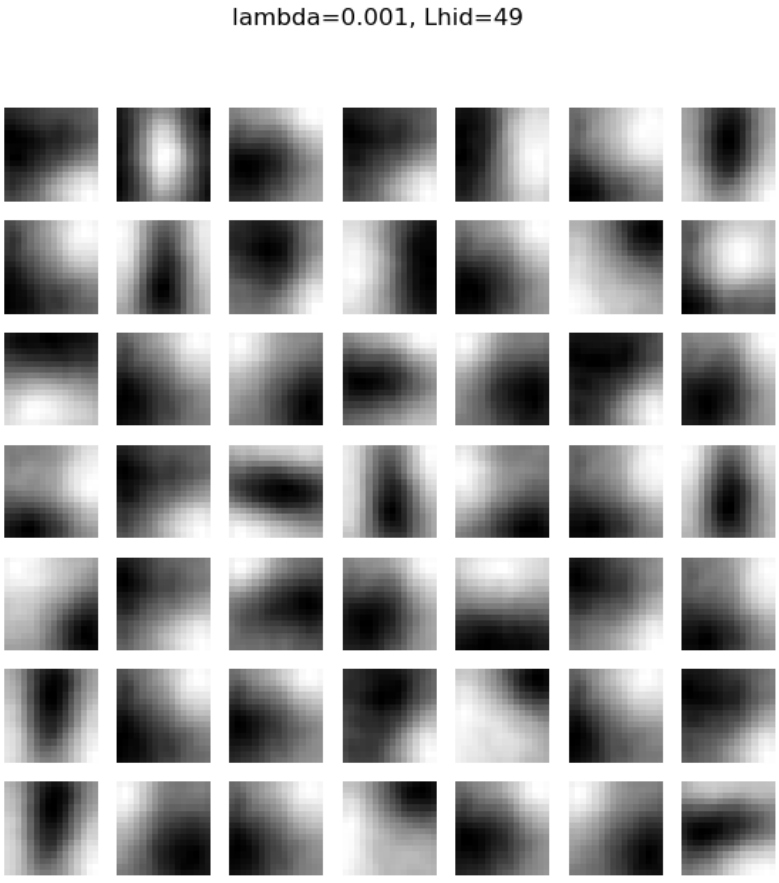


Figure 11: $\lambda = 10^{-3}, L_{hid} = 49$

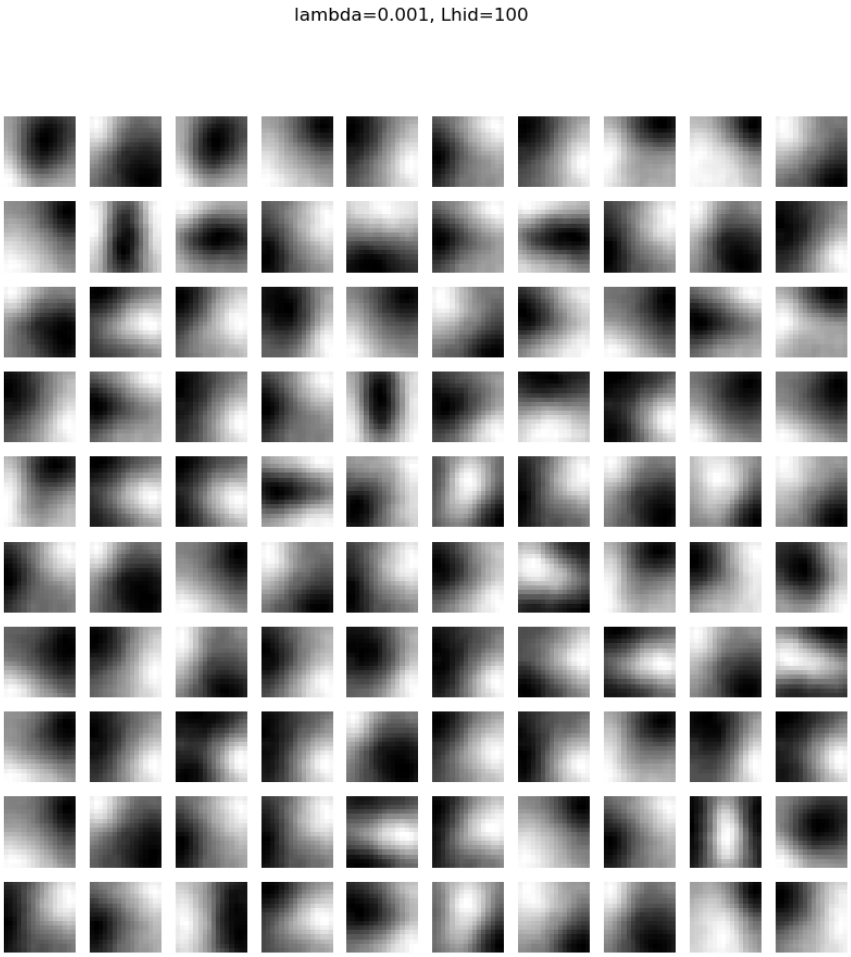


Figure 12: $\lambda = 10^{-3}, L_{hid} = 100$

Observing the results, in general, when L_{hid} is increases, different features are extracted that is a positive feature. Negatively, increasing L_{hid} very much can result in undesired and redundant features and can also result in overfitting.

For λ, increasing it can cause underfitting, and the lower values of λ cause overfitting, such as when $\lambda = 0$ since there is no regularization there is overfitting.

Therefore, λ should be not too small or large, and L_{hid} can be as large as possible but considering the much larger values, larger than 100 which is not in our interval in this case, can cause overfitting. The best combination is when $\lambda = 10^{-3}\ and\ L_{hid} = 100$. $\lambda = 10^{-3}$ is not considered as large in our case, but if it was possible to have a bigger λ, when $\lambda = 10^{-1}$ for example, there would be underfitting.

Question 2

In this question, the CNN models are introduced. One of the CNN models are introduced in Python, the other one is introduced on either TensorFlow or PyTorch frameworks. I chose the PyTorch framework. The jupyter notebook results are shown at the end of the report.

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

a) In the CNN notebook, the convolutional neural network is used on a dataset named CIFAR-10. Firstly, convolutional layer takes place in the network. The two important parameters in this first layer are padding, which reduces the loss of pixels in convolution process, and stride, which determines the windows traversal of the convolution process.

Afterwards, a cat and a dog images are shown to visualize convolution effect. After the backward pass, gradient errors are shown. Max pooling is processed, pool width and height determines the pooling window. Pooling makes the features more generalized and max pooling calculates the max value of the pooling window. Then, after the backward pass, error is displayed.

Then "Fast Layers" are introduced, which are capable of speeding up the operations by using the language C based Cython. Fast convolution and pooling layers are executed. Then, the sandwich layers, that combines the pooling, nonlinearity and convolution into one function, gave some evaluation metrics.

Then, the Three-Layer ConvNet is trained and found its accuracy nearly as 50%, and the first layer filters are displayed.

Lastly, spacial batch normalization and group normalization are mentioned. Spacial batch is a customized and shaped version of batch normalization. Group normalization is an alternative to layer normalization, to have a more effect working with CNN's.

b) As I said at the beginning, I preferred the PyTorch framework for this part. PyTorch uses tensor, which is similar to the arrays, that is used with CPUs and GPUs too, that can increase the efficiency and performence of the network. In the notebook, it creates flatten and twolayerfc functions. Using the PyTorch libraries, a two layer fully connected network is created. One of PyTorch's great advantages is it handles the intermediate values so that the gradient values do not need to be stored.

The sandwich layers in the previous CNN notebook are used to create a three layer network. Then, the two and three layered fully connected networks are trained by two initialization and accuracy calculator and training functions.

API Model of the PyTorch uses its functions for layers and replaces the previous functions in the network, that the model is used to code three layer CNN and a two layer fully connected network. Afterwards, the Sequential API is used to recreate and retrain the previous two layer fully connected network and three layer CNN. This model is very important and very convenient that can replace other training functions much easier. The results of the trained CNN and fully connected network are better than before.

Lastly, the coded model is ready, which has three convolutional models, which are merged into a PyTorch Sequential model. It has batch normalization before convolutional part and there is max pooling at the end. It is asked to get an accuracy higher than 70% by playing

with the given network. I managed to get an accuracy of The explanations and results are shown at the end of the PyTorch notebook at the end of the report.

Question 3

In this question, a human activity is classified from movement signals. The testing data contain time series of T = 150 units. The task is handled by using the first layers as RNN, LSTM and GRU layers for the following parts of the question.

Initialization is done by Xavier Uniform distribution, which is given as:

$$w_o = \sqrt{\left(\frac{6}{L_{pre} + L_{post}}\right)} \quad (5)$$

where the interval is,

$$weights, bias = [-w_o, w_o] \quad (6)$$

The forward pass and backward pass for the multilayer perceptron are shown below:

$$h = \phi(x.weights + bias) \quad (7)$$

$$\frac{dC}{dweights} = h^T.\delta \quad (8)$$

$$\frac{dC}{dbias} = [1]_{1\times L}.\delta \quad (9)$$

Since we are using RNN, LSTM and GRU for this question, δ error gradient is updated at each layer.

For activations, hidden multi-layer perceptrons use ReLU, output layer uses sigmoid function. Cross-entropy error function is used for errors. For backpropogation, BPTT is used for the three different RNN, LSTM and GRU layers. Additionally, momentum is used to get better and more accurate results. The update equation and the update after momentum equation are shown below:

$$\Delta weights = \sum_{t=t_0}^{T} \frac{dC(t)}{dweights} \quad (10)$$

$$weights \ -= \ \eta.\Delta weights + \alpha.weights_{momentum}$$

I wrote the code to save my results in images that will save them in the same folder with the code script.

a) In this part, network is trained by using the first layer as Recurrent layer.

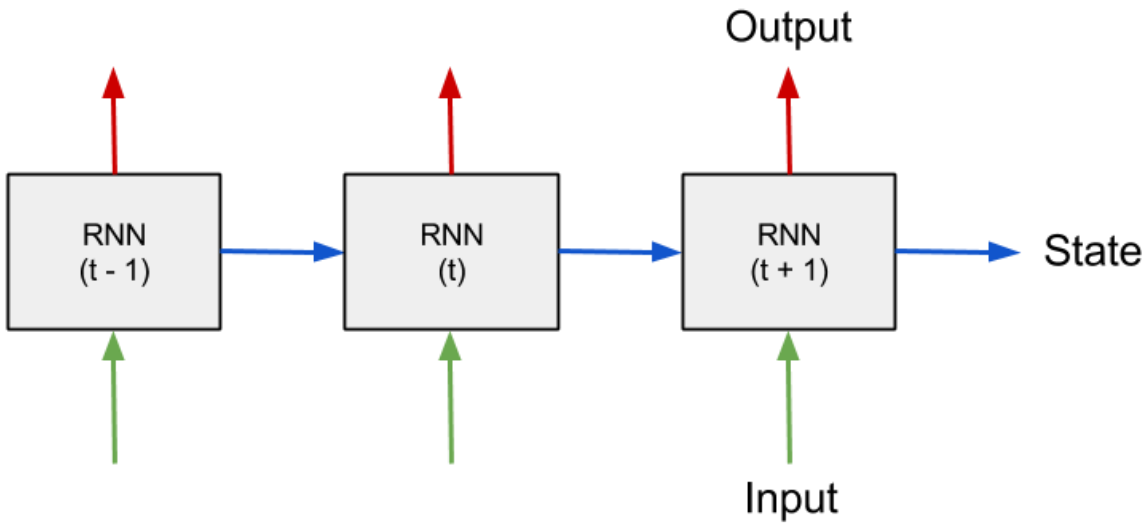


Figure 13: Recurrent layer architecture

RNN's are like the multi-layer perceptron with some changes. It is an extension of multi-layer perceptron that takes the input and also the past output as an input. The respective derivation for our network can be shown as:

$$h(t) = \tanh(x(t) * weights_{ih} + h(t-1) * weights_{hh} + bias) \quad (11)$$

The results are shown below:

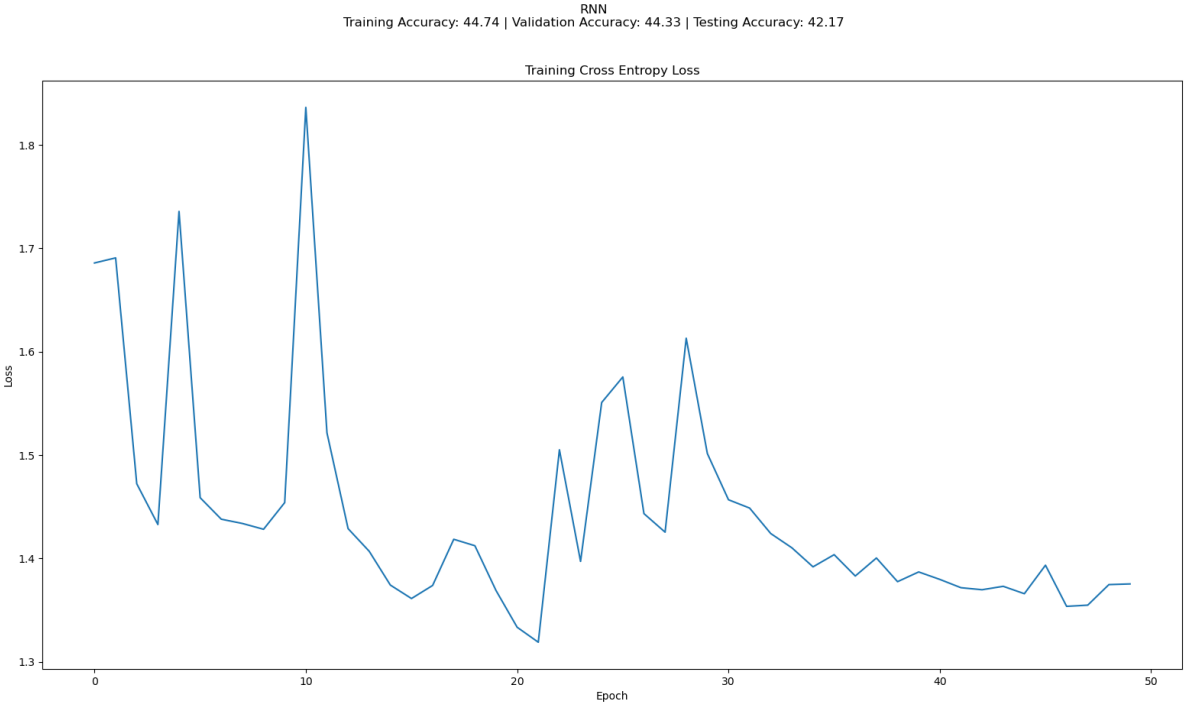


Figure 14: RNN Training Cross Entropy Loss

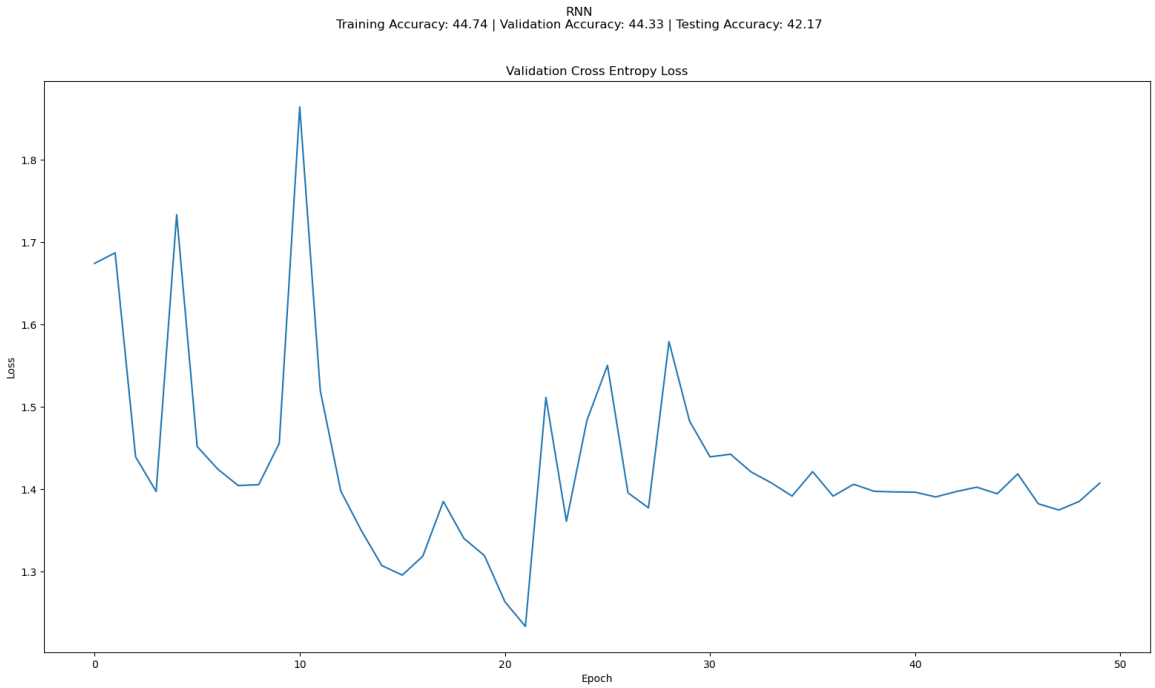


Figure 15: RNN Validation Cross Entropy Loss

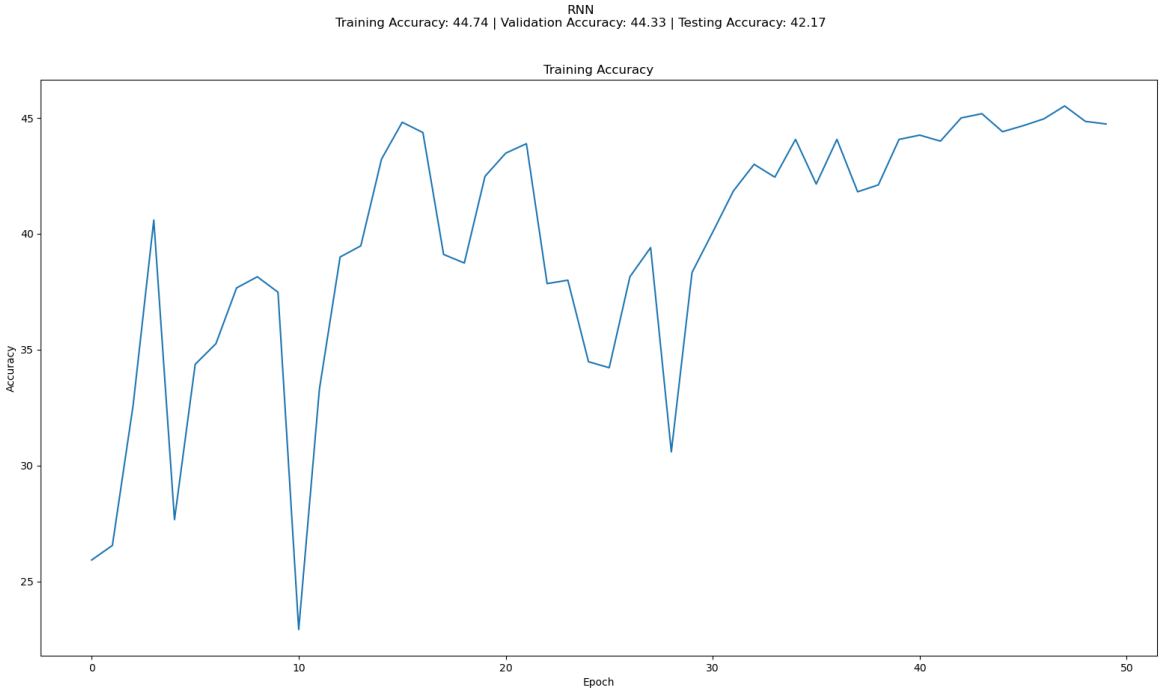


Figure 16: RNN Training Accuracy

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

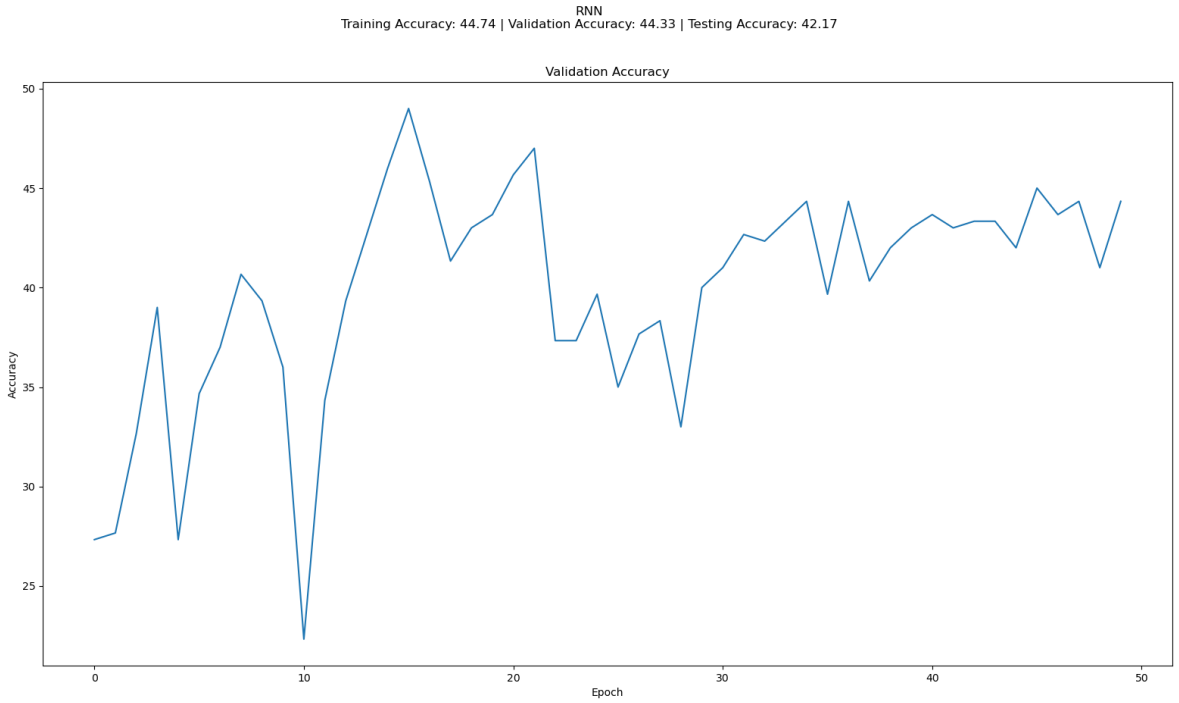


Figure 17: RNN Validation Accuracy

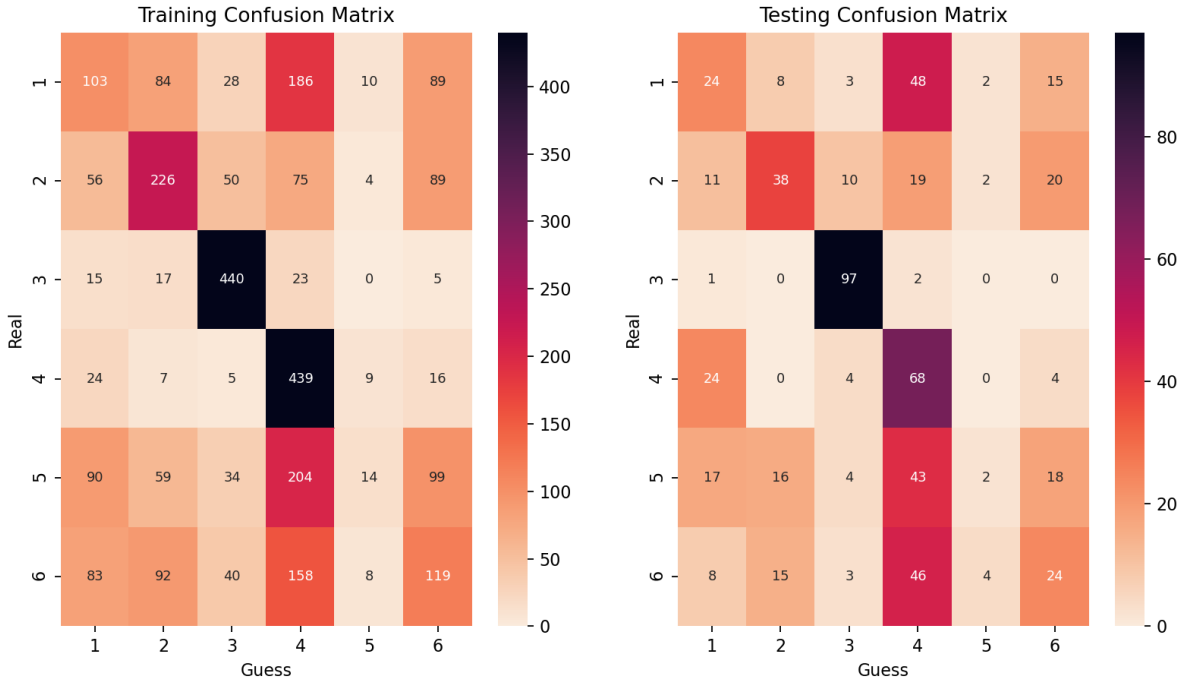


Figure 18: RNN Confusion Matrixes

Training with RNN, there occur some unstability in the loss and accuracy plots. Due to vanishing and exploding gradients problem of RNN, in the long run, it does not always

approaches to a smaller loss when the training goes on, then the network is not learning. Gradients become very high or very low in this case, that results in rapid changes in loss. Having 150 time samples and the cumulative features of the BPTT also results in rapid changes in gradient for RNN.

The parameters can effect the unstability. For example, decreasing the learning rate may decrease the overshooting, but may also result in slow training or even stopping the learning process if it is decreased too much. LSTM and GRU aim to overcome these problems, which will be discussed in the following parts.

b) In this part, network is trained by using the first layer as LSTM.

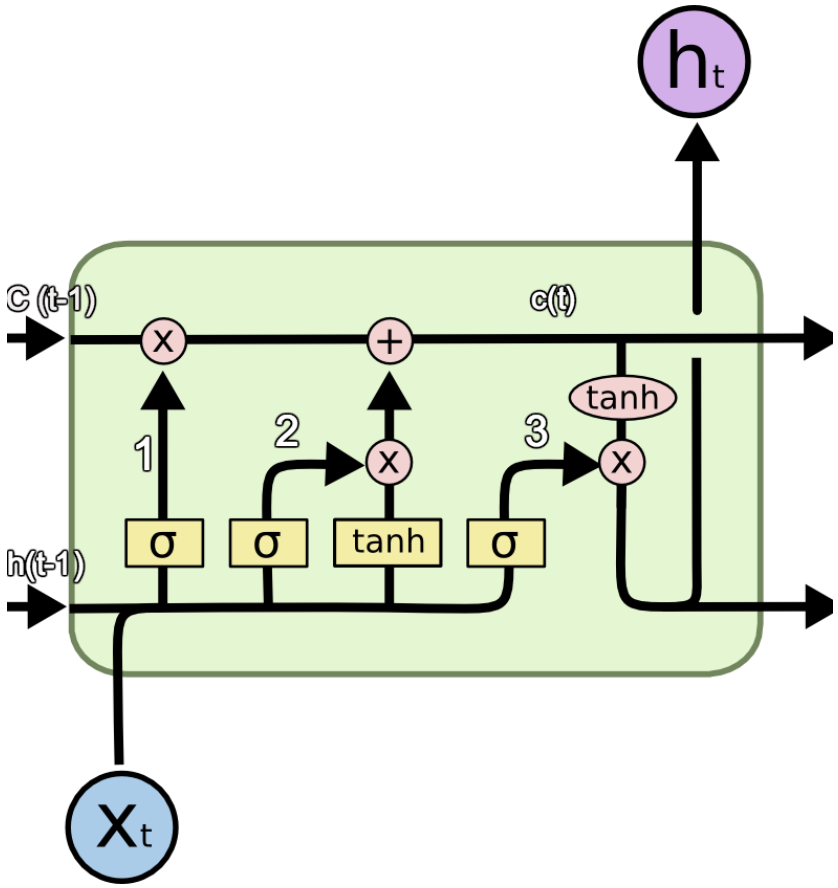


Figure 19: LSTM layer architecture

LSTM solves the problem that occurs in RNN, which is exploding or vanishing gradients. It uses forget gates to forget some of the past memory in the network, which increases the stability. In our network, the equations of LSTM are:

$$hi(t) = \sigma([h(t-1), x(t)].weights_i + bias_i) \quad (12)$$

$$hf(t) = \sigma\big([h(t-1), x(t)].weights_f + bias_f\big) \quad (13)$$

$$ho(t) = \sigma([h(t-1), x(t)].weights_o + bias_o) \quad (14)$$

$$hc(t) = tanh([h(t-1), x(t)].weights_c + bias_c) \quad (15)$$

$$c(t) = hf(t).c(t-1) + hi(t).hc(t) \quad (16)$$

$$h(t) = ho(t).\tanh\big(c(t)\big) \quad (17)$$

The results are shown below:

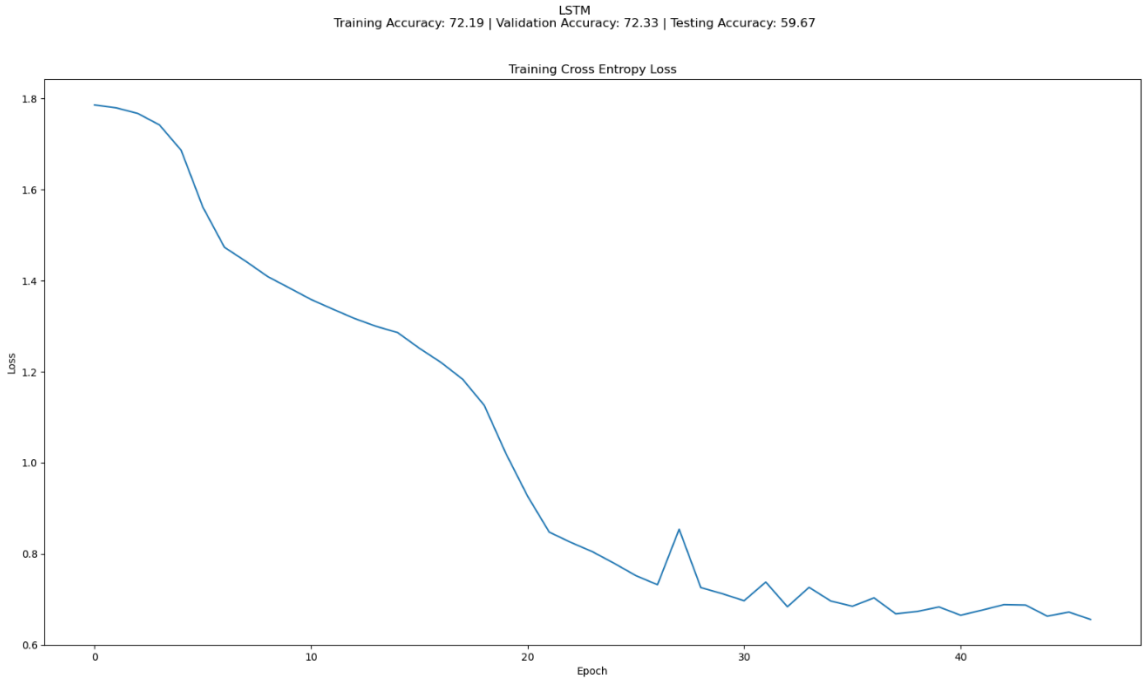


Figure 20: LSTM Training Cross Entropy Loss

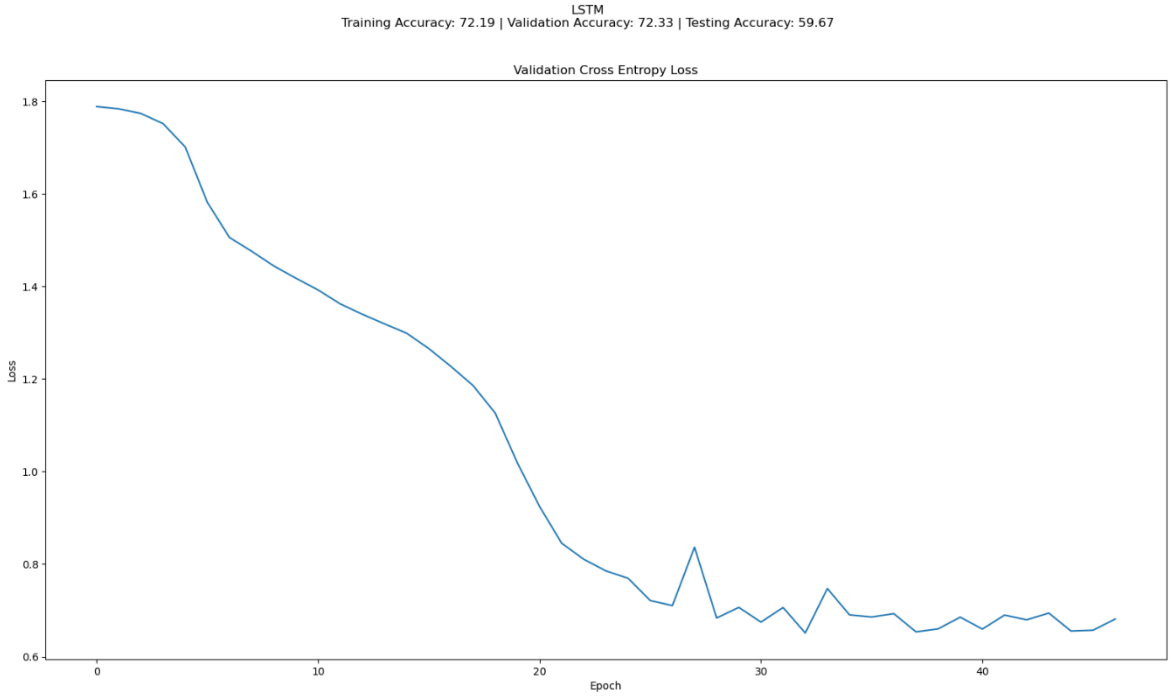


Figure 21: LSTM Validation Cross Entropy Loss

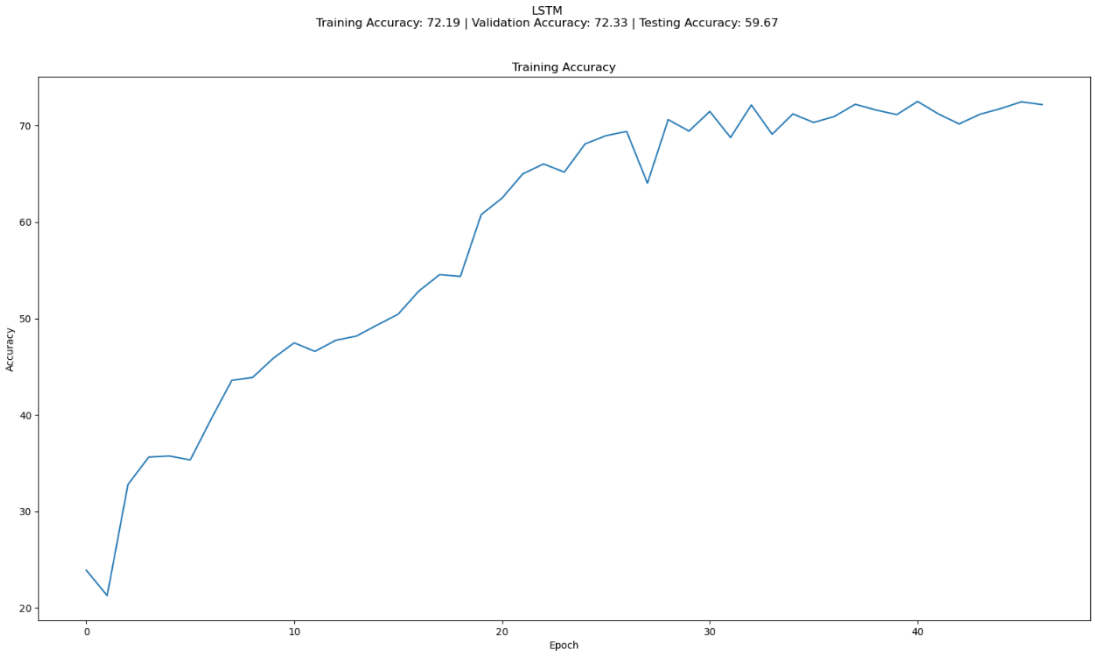


Figure 22: LSTM Training Accuracy

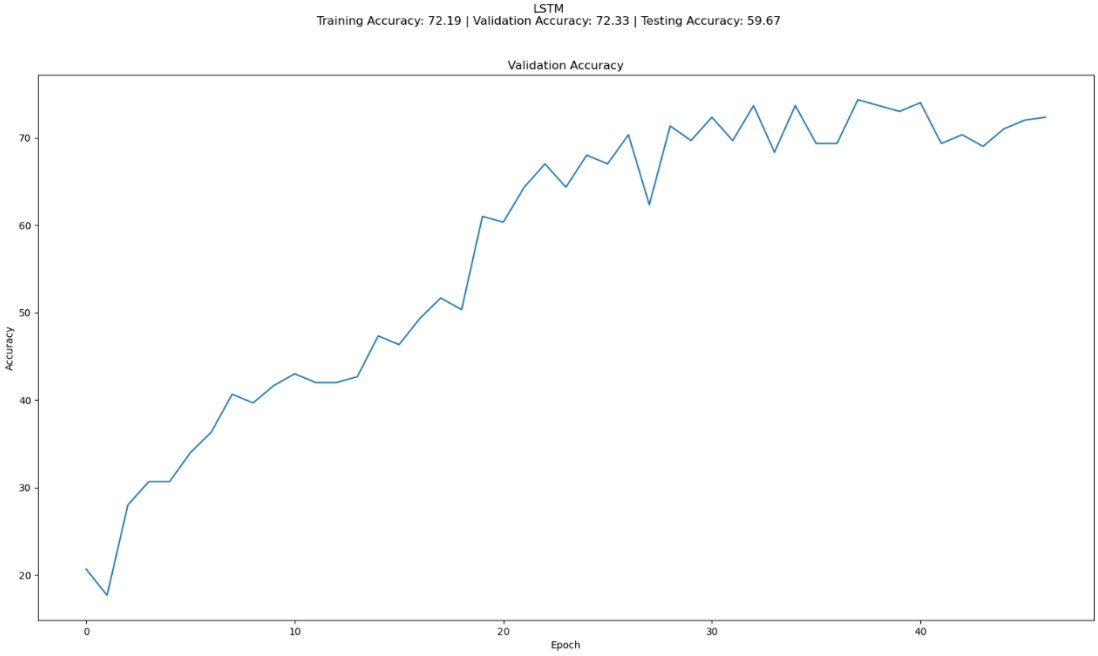


Figure 23: LSTM Validation Accuracy

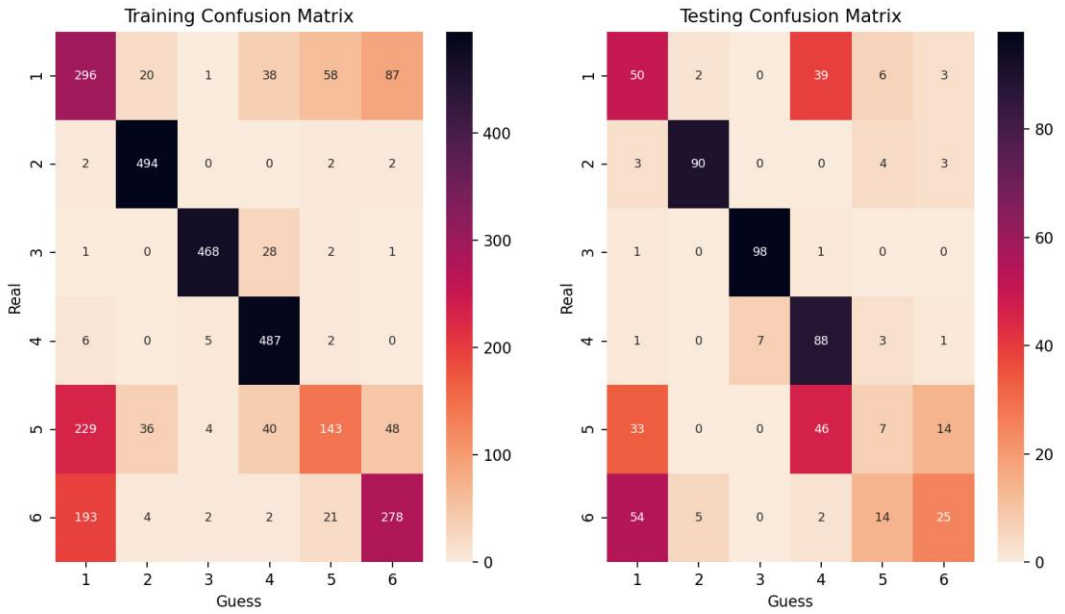


Figure 24: LSTM Confusion Matrixes

Compared to RNN, the network accuracy has increased greatly. Confusion matrices and the test accuracy shows that networks learns and capable of doing correct predictions. Test accuracy becomes 59.67%, where the test accuracy was 42.17% in RNN. Although accuracy has increased, it can be increased more by increasing the learning rate, but it should be fixed in our problem. However, increasing the learning rate too much can also result in exploding gradients.

c) In this last part, network is trained by using the first layer as GRU.

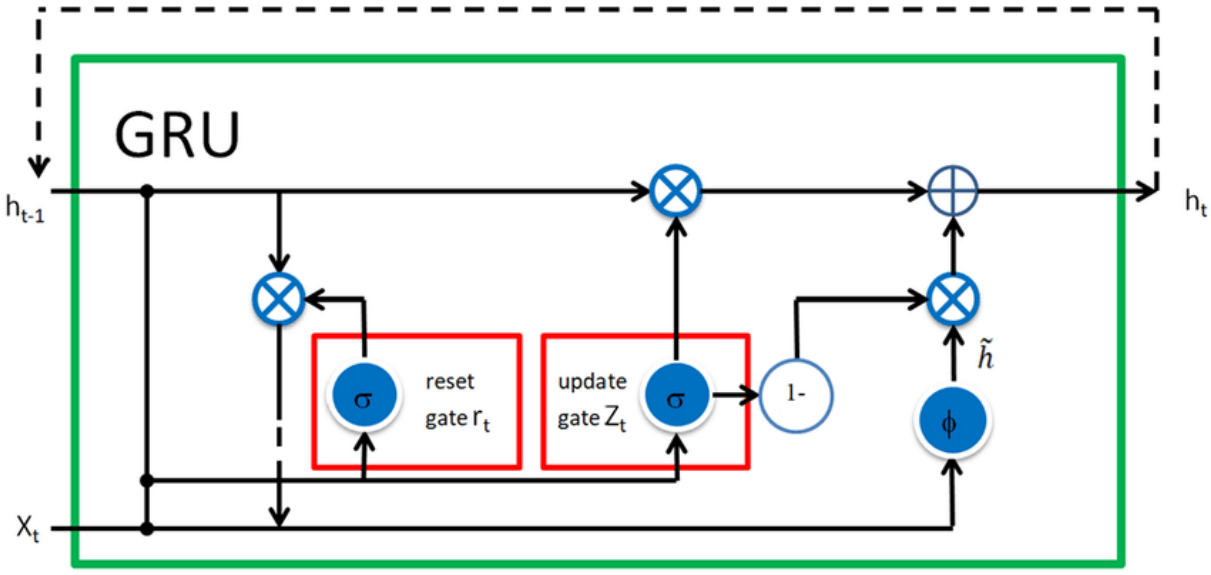


Figure 25: GRU layer architecture

GRU is similar to the LSTM network with slight changes, it has less complexity which is a result of less gates and equations included in it. However, the performance is still great and even better than LSTM that I will talk about after the results. The respective equations for our network are:

$$z(t) = \sigma(x(t).weights_z + h(t-1) * u_z + bias_z) \quad (18)$$

$$r(t) = \sigma(x(t).weigts_r + h(t-1) * u_r + bias_r) \quad (19)$$

$$\tilde{h}(t) = \tanh\big(x(t).weights_h + \big(h(t-1).r(t)\big)u_h + bias_h\big) \quad (20)$$

$$h(t) = \big(1 - z(t)\big).h(t-1) + z(t).\tilde{h}(t) \quad (21)$$

The results are shown below:

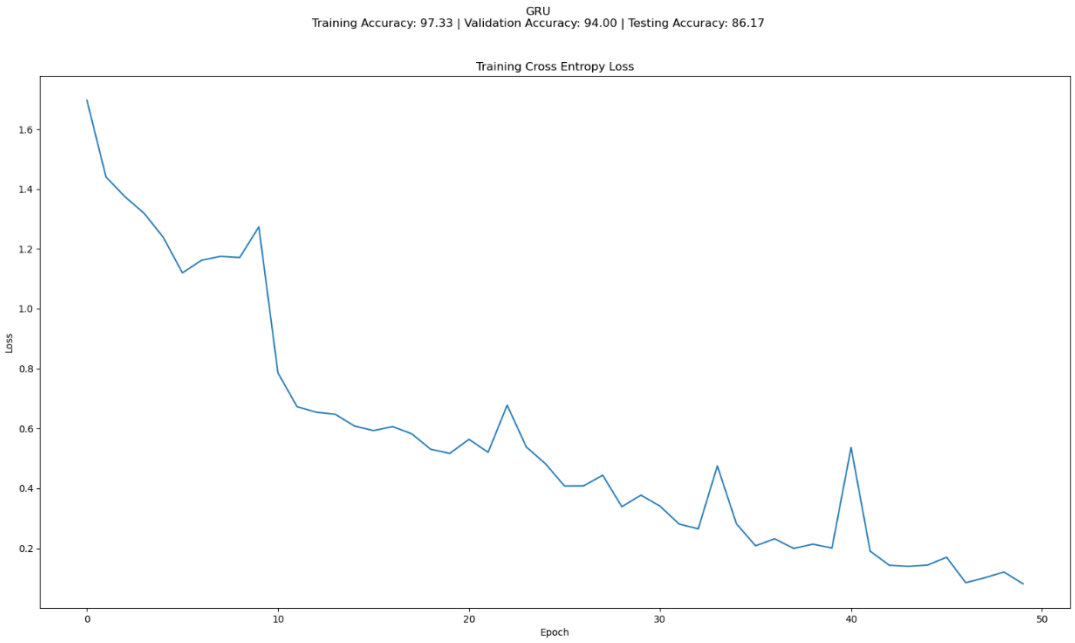


Figure 26: GRU Training Cross Entropy Loss

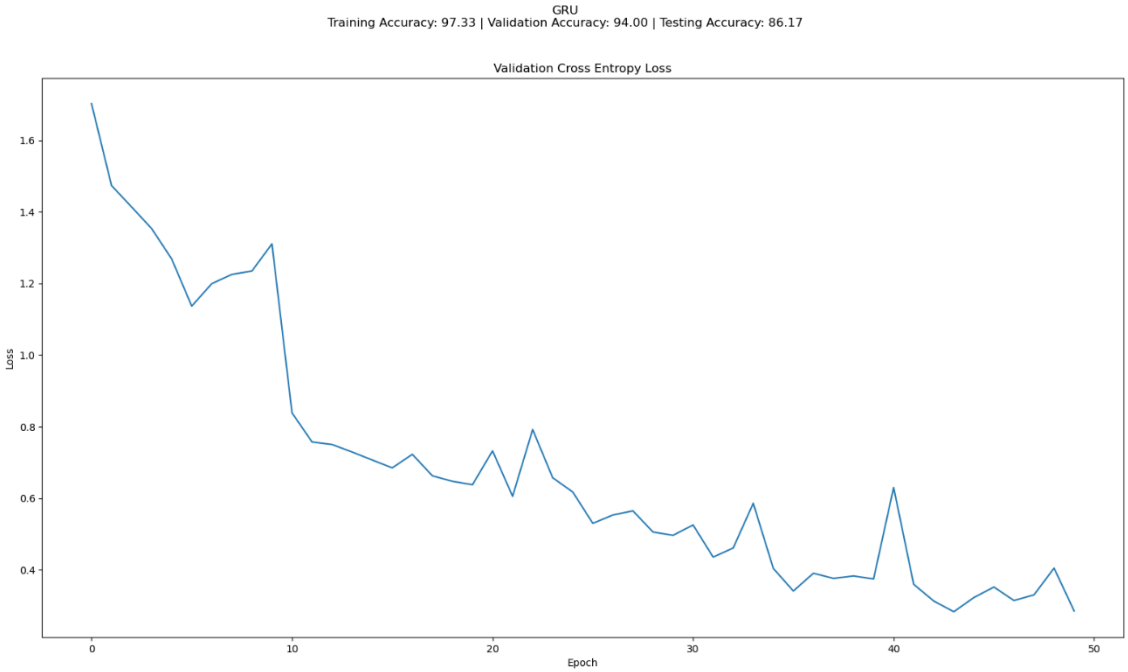


Figure 27: GRU Validation Cross Entropy Loss

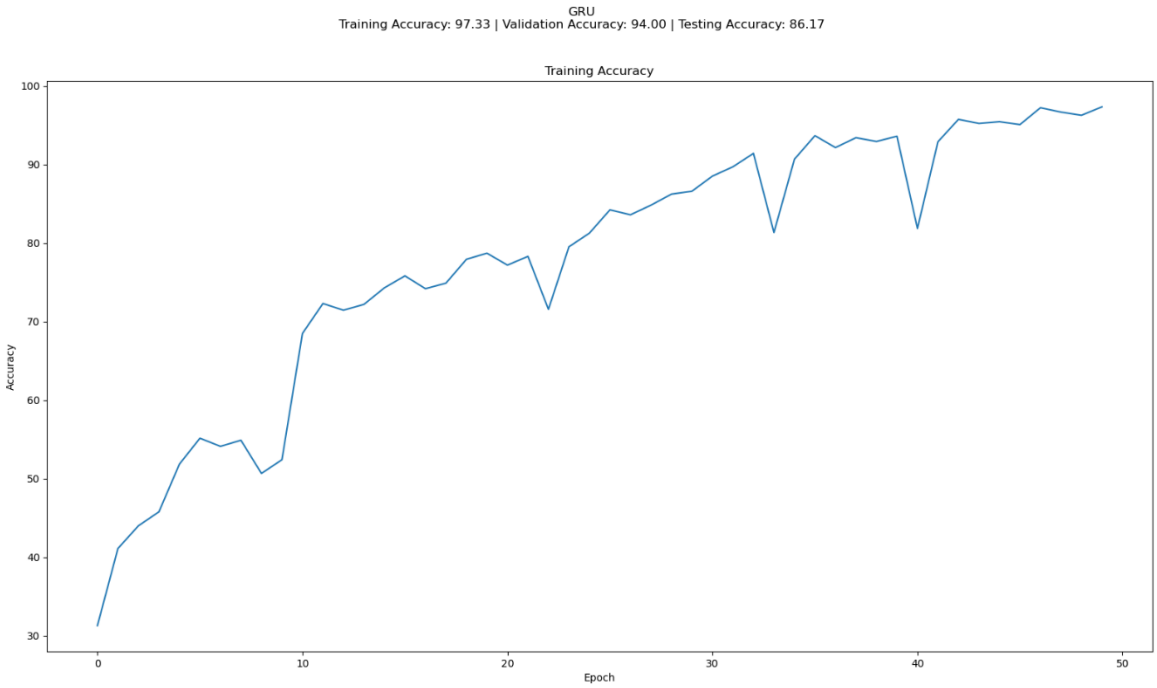


Figure 28: GRU Training Accuracy

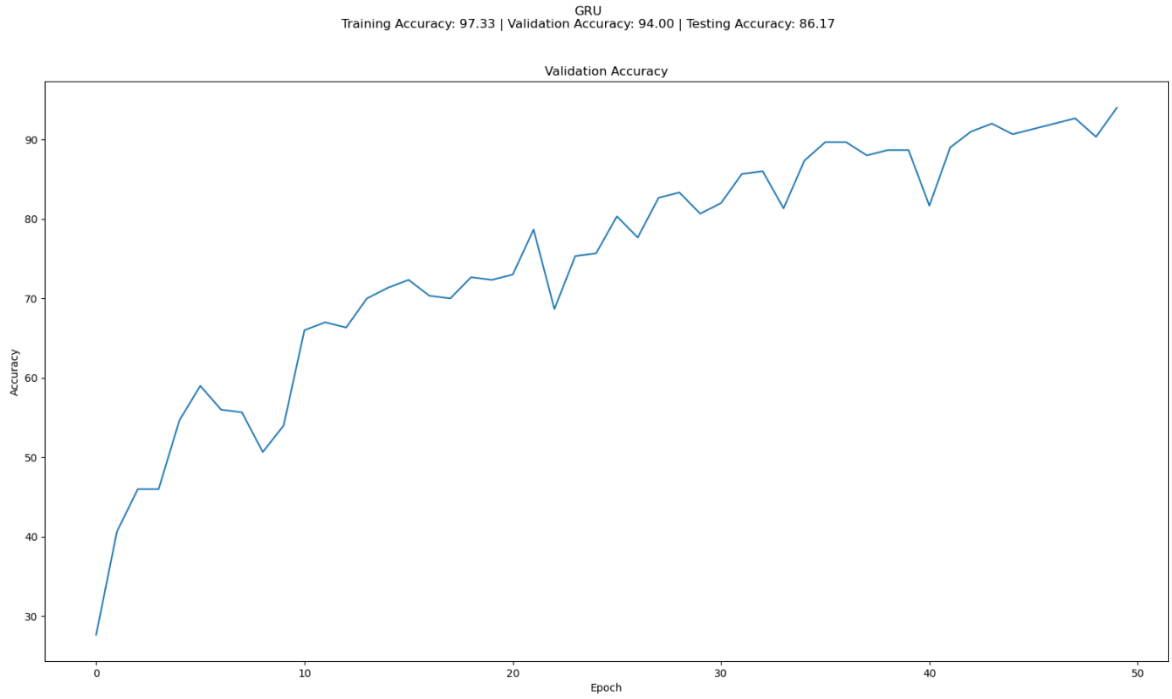


Figure 29: GRU Validation Accuracy

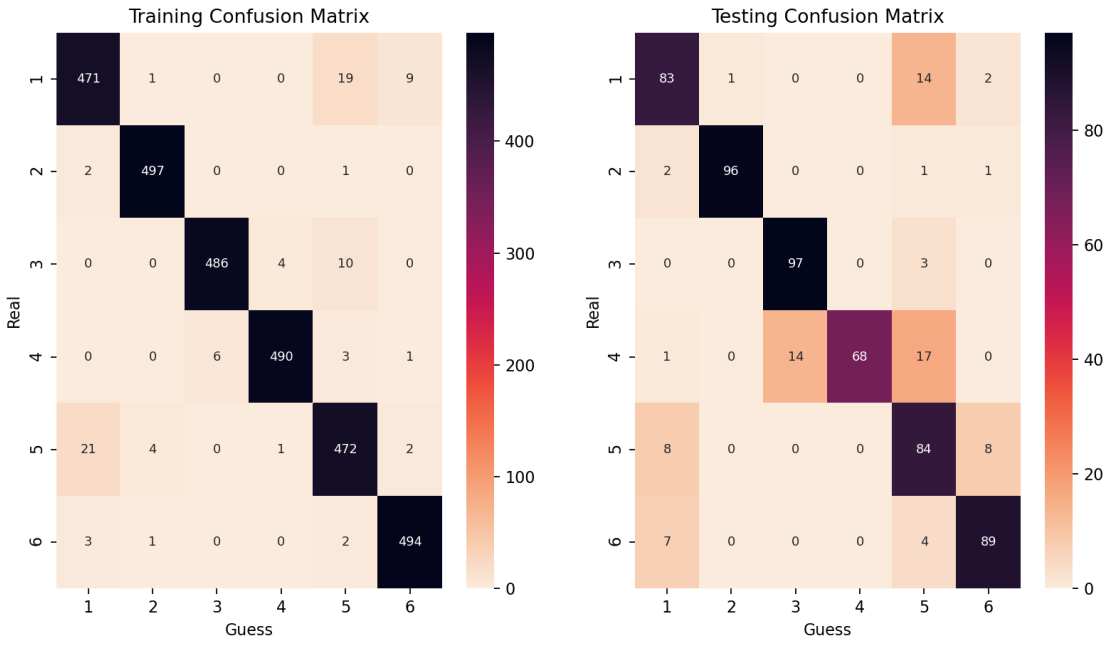


Figure 30: GRU Confusion Matrixes

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

Compared to the other networks, GRU has the highest accuracy and highest performence. It has a test accuracy of 86.17%. It also had a stable loss at the end, which means that it also solves the problem of vanishing or exploding gradients. Compared to LSTM, GRU's training time is less since it has less gates inside. However, LSTM is more stable and less affected by the gradients' vanishing or exploding. The reason behind this is GRU does not entirely gets rid of the vanishing gradients, just decrease them. LSTM overcomes this situation better, therefore it becomes more stable. Therefore, GRU is built to be efficient, and LSTM is built to have a more stable and accurate network.

Ayberk Yarkın Yıldız
21803386

Appendix

Script That is Run in Command Prompt via "python ayberk_yarkin_yildiz_21803386_hw3.py x", For Both Question 1 and Question 3

```
import sys
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sn
import h5py

def getting(n, bi, tr1, tr2, l, m, b, e, te1, te2, no):
    net = n(bi,no)
    trloss, valoss, tracc, valacc = net.workout3(tr1,tr2,l,m,b,e).values()
    testacc = net.guess3(te1, te2, accur=True)
    return net, trloss, valoss, tracc, valacc, testacc

def guessing(n, tr1, tr2, te1, te2):
    trconf = n.guess3(tr1, tr2, accur=True, conf=True)
    teconf = n.guess3(te1, te2, accur=True, conf=True)
    return trconf, teconf

def lstmproc(w, b, zic, k):
    w = w + zic.T @ k
    b = b + k.sum(axis=0, keepdims=True)
    return w,b

def lstmdu(k, w, big):
    f = k @ w.T[:, :big]
```

```
    return f


def gruproc(a, d, h, w, u, b):
        w = w + a.T @ d
        u = u + h.T @ d
        b = b + d.sum(axis=0, keepdims=True)
        return w,u,b


def ifing1(h, b, n, k):
    if k > 0:
        hb = h[:, k-1, :]
    else:
        hb = np.zeros((n,b))
    return hb


def ifing2(h, k):
    if k > 0:
        hb = h[:, k-1, :]
    else:
        hb = 0
    return hb


def graphn(tr, val, te, dat, namee):
    fig = plt.figure(figsize=(20, 10))
    fig.suptitle(str(namee)+"\nTraining Accuracy: {:.2f} | Validation Accuracy: {:.2f} | Testing
Accuracy: {:.2f}\n ".format(tr[-1], val[-1], te))
    plt.plot(dat)
    plt.xlabel("Epoch")
```

```
def graphm(trconf, testconf):
    plt.figure(figsize=(20, 10), dpi=160)
    plt.subplot(1, 2, 1)
    sn.heatmap(trconf, annot=True, annot_kws={"size": 8}, xticklabels=[1, 2, 3, 4, 5, 6], yticklabels=[1, 2, 3, 4, 5, 6], cmap=sn.cm.rocket_r, fmt='g')
    plt.title("Training Confusion Matrix")
    plt.ylabel("Real")
    plt.xlabel("Guess")
    plt.subplot(1, 2, 2)
    sn.heatmap(testconf, annot=True, annot_kws={"size": 8}, xticklabels=[1, 2, 3, 4, 5, 6], yticklabels=[1, 2, 3, 4, 5, 6], cmap=sn.cm.rocket_r, fmt='g')
    plt.title("Testing Confusion Matrix")
    plt.ylabel("Real")
    plt.xlabel("Guess")


def calcul(a,b,f,k):
    accu = f(a,b,accur=True)
    loso = k(b, f(a,accur=False))
    return accu, loso


def norma(x):
    return (x-x.min())/(x.max()-x.min())


def dis(lrate, mom, epoch, batch, rho, beta, lamda, Lin, Lhid, x, t, d):
    params = {"rho": rho, "beta": beta, "lamda": lamda, "Lin": Lin, "Lhid": Lhid}
    autoencoder = x()
    wson = norma(autoencoder.workout1(t, params, lrate, mom, epoch, batch)[0][0]).T
    wson = wson.reshape((wson.shape[0], d, d))
    w_dimension = int(np.sqrt(wson.shape[0]))
```

```
    return w_dimension, wson


def imageshowc(w,d,lamda,Lhid):
    fig, ax = plt.subplots(d, d, figsize=(d, d))
    fig.suptitle("Question 1, Part c, lambda={}, Lhid={}".format(lamda, Lhid))
    c = 0
    for a in range(d):
        for b in range(d):
            ax[a,b].imshow(w[c], cmap='gray')
            ax[a,b].axis('off')
            c = c + 1
    plt.show()


def imageshowd(w,d,lamda,Lhid):
    fig, ax = plt.subplots(d, d, figsize=(d, d))
    fig.suptitle("Question 1, Part d, lambda={}, Lhid={}".format(lamda, Lhid))
    c = 0
    for a in range(d):
        for b in range(d):
            ax[a,b].imshow(w[c], cmap='gray')
            ax[a,b].axis('off')
            c = c + 1
    plt.show()


def q1():


    class AE():
        def initialize1(self, Lin, Lhid):
            Lout = Lin
```

```
        W1 = np.random.uniform(-(np.sqrt(6)/np.sqrt(Lin + Lhid)),(np.sqrt(6)/np.sqrt(Lin + Lhid)), size=(Lin,Lhid))

        b1 = np.random.uniform(-(np.sqrt(6)/np.sqrt(Lin + Lhid)),(np.sqrt(6)/np.sqrt(Lin + Lhid)), size=(1, Lhid))


        W2 = W1.T

        b2 = np.random.uniform(-np.sqrt(6)/np.sqrt(Lhid + Lout),np.sqrt(6)/np.sqrt(Lhid + Lout), size=(1,Lout))


        We = (W1, W2, b1, b2)

        momWe = (0,0,0,0)


        return We, momWe


    def solver(self, Jgrad, co, We, momWe, lrate, mom):


        W1, W2, b1, b2 = We

        derW1, derW2 = 0,0


        data, h, hder, oder = co

        derloss, dertako2, dertako1, derkel = Jgrad


        derW2 = h.T @ (derloss * oder)+dertako2

        derW1 = data.T @ (hder*((derloss * oder) @ W2.T + derkel)) + dertako1

        derWe = (((derW1.T + derW2)/2).T, (derW1.T + derW2)/2, (hder*((derloss * oder) @ W2.T + derkel)).sum(axis=0, keepdims=True), (derloss * oder).sum(axis=0, keepdims=True))


        We, momWe = self.modify(We, momWe, derWe, lrate, mom)


        return We, momWe
```

```
    def modify(self, We, momWe, derWe, lrate, mom):


        W1, W2, b1, b2 = We

        derW1, derW2, derb1, derb2 = derWe

        momW1, momW2, momb1, momb2 = momWe


        assert(W1 == W2.T).all()

        We = (W1 - (lrate*derW1 + mom*momW1),W2 - (lrate*derW2 + mom*momW2),b1 -
(lrate*derb1 + mom*momb1),b2 - (lrate*derb2 + mom*momb2))

        momWe = (lrate*derW1 + mom*momW1,lrate*derW2 + mom*momW2,lrate*derb1
+ mom*momb1,lrate*derb2 + mom*momb2)


        return We, momWe


    def workout1(self, data, params, lrate, mom, epoch, batch):

        Jl = []


        Lin = params["Lin"]

        Lhid = params["Lhid"]

        We, momWe = self.initialize1(Lin, Lhid)

        for i in range(epoch):


            Jt = 0

            go = 0

            stop = batch


            data = data[np.random.permutation(data.shape[0])]
```

```
        momWe = (0,0,0,0)

        for j in range(int(data.shape[0]/batch)):
            batching = data[go:stop]

            J, Jgrad, co = self.aeCost(We, batching, params)
            We, momWe = self.solver(Jgrad, co, We, momWe, lrate, mom)

            Jt = Jt + J
            go = stop
            stop = stop + batch

        Jt = Jt/(int(data.shape[0]/batch))

        print("Epoch: {}, Loss: {:.3f}".format(i+1, Jt))
        Jl.append(Jt)

    return We, Jl

def aeCost(self, We, data, params):

    nar = data.shape[0]

    W1, W2, b1, b2 = We

    rho = params["rho"]
    beta = params["beta"]
    lamda = params["lamda"]
    Lin = params["Lin"]
```

```
        Lhid = params["Lhid"]


        h, hder = self.sigmoid(data @ W1+b1)


        o, oder = self.sigmoid(h @ W2+b2)


        rhobe = h.mean(axis=0, keepdims=True)


        J = ((0.5*(np.linalg.norm(data-o,axis=1)**2).sum())/nar) +
(0.5*lamda*(np.sum(W1**2) + np.sum(W2**2))) + (beta * (rho*np.log(rho/rhobe) + (1-
rho)*np.log((1-rho)/(1-rhobe))).sum())


        co = (data, h, hder, oder)

        Jgrad = ((o-data)/nar, lamda * W2, lamda * W1, (beta * ((1-rho)/(1-rhobe) -
rho/rhobe)/nar))


        return J, Jgrad, co


    def sigmoid(self, x):

        k = np.exp(x)/(1 + np.exp(x))

        l = k * (1-k)

        return k,l


filename1 = "assign3_data1.h5"

f1 = h5py.File(filename1, 'r')

data = np.array(f1['data'])


""" Part a """

datanew = 0.2126 * data[:, 0] + 0.7152 * data[:, 1] + 0.0722 * data[:, 2]
```

```
    assert datanew.shape[1] == datanew.shape[2]

    dimension = datanew.shape[1]


    datanew = (np.reshape(datanew, (datanew.shape[0], dimension ** 2))) -
(np.reshape(datanew, (datanew.shape[0], dimension ** 2))).mean(axis=1, keepdims = True)

    datanew = 0.1 + 0.8*(norma(np.clip(datanew, - 3 * (np.std(datanew)), 3 *
(np.std(datanew)))))


    trd = datanew


    datanew = np.reshape(datanew, (datanew.shape[0], dimension, dimension))

    data = data.transpose((0,2,3,1))


    """ Part c """


    print("\nQuestion 1 Part c")

    epoch = 200

    mom = 0.8

    rho = 0.03

    beta = 3

    lrate = 0.1

    batch = 32

    lamda = 5e-4

    Lin = trd.shape[1]

    Lhid = 64

    print("\nParameters: rho =",rho,",","beta =",beta,",","lrate =",lrate,",","momentum
=",mom,",","lambda =",lamda,",","batch =",batch,",","Lin =",Lin,",","Lhid =",Lhid,",","epoch
=",epoch,"\n")

    w_dimensioni, wsoni = dis(lrate, mom, epoch, batch, rho, beta, lamda, Lin, Lhid, AE, trd,
dimension)
```

```
""" Part d """


print("\nQuestion 1 Part d")

Lhidl = 16

Lhidn = 49

Lhidh = 100

lamdal = 0

lamdan = 1e-5

lamdah = 1e-3


print("\nlambda =",lamdal,","Lhid =",Lhidl)

w_dimension1, wson1 = dis(lrate, mom, epoch, batch, rho, beta, lamdal, Lin, Lhidl, AE, trd, dimension)

print("\nlambda =",lamdal,","Lhid =",Lhidn)

w_dimension2, wson2 = dis(lrate, mom, epoch, batch, rho, beta, lamdal, Lin, Lhidn, AE, trd, dimension)

print("\nlambda =",lamdal,","Lhid =",Lhidh)

w_dimension3, wson3 = dis(lrate, mom, epoch, batch, rho, beta, lamdal, Lin, Lhidh, AE, trd, dimension)

print("\nlambda =",lamdan,","Lhid =",Lhidl)

w_dimension4, wson4 = dis(lrate, mom, epoch, batch, rho, beta, lamdan, Lin, Lhidl, AE, trd, dimension)

print("\nlambda =",lamdan,","Lhid =",Lhidn)

w_dimension5, wson5 = dis(lrate, mom, epoch, batch, rho, beta, lamdan, Lin, Lhidn, AE, trd, dimension)

print("\nlambda =",lamdan,","Lhid =",Lhidh)

w_dimension6, wson6 = dis(lrate, mom, epoch, batch, rho, beta, lamdan, Lin, Lhidh, AE, trd, dimension)

print("\nlambda =",lamdah,","Lhid =",Lhidl)
```

```
    w_dimension7, wson7 = dis(lrate, mom, epoch, batch, rho, beta, lamdah, Lin, Lhidl, AE,
trd, dimension)
    print("\nlambda =",lamdah,",","Lhid =",Lhidn)
    w_dimension8, wson8 = dis(lrate, mom, epoch, batch, rho, beta, lamdah, Lin, Lhidn, AE,
trd, dimension)
    print("\nlambda =",lamdah,",","Lhid =",Lhidh)
    w_dimension9, wson9 = dis(lrate, mom, epoch, batch, rho, beta, lamdah, Lin, Lhidh, AE,
trd, dimension)



    """Plots"""
    print('\nTo continue executing, after observation please close the plots')
    print("\nQuestion 1 Part a Plot")
    fig1, ax1 = plt.subplots(10,20,figsize=(20,10))
    fig1.suptitle("Question 1, Part a, RGB Images")
    fig2, ax2 = plt.subplots(10, 20, figsize=(20, 10))
    fig2.suptitle("Question 1, Part a, Grayscale Images")

    for a in range (10):
        for b in range (20):
            c = np.random.randint(0, data.shape[0])
            ax2[a,b].imshow(datanew[c], cmap='gray')
            ax2[a,b].axis("off")

            ax1[a,b].imshow(data[c].astype('float'))
            ax1[a,b].axis('off')

    print('\nTo continue executing, after observation please close the plots')
    plt.show()
```

```
    print("\nQuestion 1 Part c Plot")
    imageshowc(wsoni, w_dimensioni, lamda, Lhid)
    print("\nQuestion 1 Part d Plots")
    imageshowd(wson1, w_dimension1, lamdal, Lhidl)
    imageshowd(wson2, w_dimension2, lamdal, Lhidn)
    imageshowd(wson3, w_dimension3, lamdal, Lhidh)
    imageshowd(wson4, w_dimension4, lamdan, Lhidl)
    imageshowd(wson5, w_dimension5, lamdan, Lhidn)
    imageshowd(wson6, w_dimension6, lamdan, Lhidh)
    imageshowd(wson7, w_dimension7, lamdah, Lhidl)
    imageshowd(wson8, w_dimension8, lamdah, Lhidn)
    imageshowd(wson9, w_dimension9, lamdah, Lhidh)


def q3():
    class Network3():

        def __init__(self,big,num):

            self.num = num
            self.big = big
            self.laybig = len(big)-1
            self.percbig = None
            self.percpar = None
            self.flaypar = None
            self.percmom = None
            self.flaymom = None
            self.initialize3()
```

```
    def initialize3(self):
        num = self.num
        big = self.big
        laybig = self.laybig

        weights = []
        bias = []

        for i in range(1,laybig):
            weights.append(np.random.uniform(-(np.sqrt(6)/np.sqrt(big[1] +
big[i+1])),(np.sqrt(6)/np.sqrt(big[1] + big[i+1])), size=(big[i],big[i+1])))
            bias.append(np.zeros((1, big[i+1])))

        self.percbig = len(weights)
        params = {"weights":weights, "bias":bias}
        mom = {"weights": [0]*self.percbig, "bias": [0]*self.percbig}
        self.percpar = params
        self.percmom = mom

        ne = big[0]
        he = big[1]
        ze = ne + he

        if num == 1:
            weightsih = np.random.uniform(-(np.sqrt(6)/np.sqrt(ne+he)),
(np.sqrt(6)/np.sqrt(ne+he)), size = (ne,he))
            weightshh = np.random.uniform(-(np.sqrt(6)/np.sqrt(he+he)),
(np.sqrt(6)/np.sqrt(he+he)), size = (he,he))
            bias = np.zeros((1,he))
```

```
        params = {"weightsih": weightsih, "weightshh": weightshh, "bias": bias}


    if num == 2:

        weightsf = np.random.uniform(-(np.sqrt(6)/np.sqrt(ze+he)),
(np.sqrt(6)/np.sqrt(ze+he)), size = (ze,he))

        biasf = np.zeros((1,he))

        weightsi = np.random.uniform(-(np.sqrt(6)/np.sqrt(ze+he)),
(np.sqrt(6)/np.sqrt(ze+he)), size = (ze,he))

        biasi = np.zeros((1,he))

        weightsc = np.random.uniform(-(np.sqrt(6)/np.sqrt(ze+he)),
(np.sqrt(6)/np.sqrt(ze+he)), size = (ze,he))

        biasc = np.zeros((1,he))

        weightso = np.random.uniform(-(np.sqrt(6)/np.sqrt(ze+he)),
(np.sqrt(6)/np.sqrt(ze+he)), size = (ze,he))

        biaso = np.zeros((1,he))


        params = {"weightsf": weightsf, "biasf": biasf,"weightsi": weightsi, "biasi":
biasi,"weightsc": weightsc, "biasc": biasc,"weightso": weightso, "biaso": biaso}


    if num == 3:

        weightsz = np.random.uniform(-(np.sqrt(6)/np.sqrt(ne+he)),
(np.sqrt(6)/np.sqrt(ne+he)), size=(ne, he))

        uzaz = np.random.uniform(-(np.sqrt(6)/np.sqrt(he+he)),
(np.sqrt(6)/np.sqrt(he+he)), size=(he, he))

        biasz = np.zeros((1, he))

        weightsr = np.random.uniform(-(np.sqrt(6)/np.sqrt(ne+he)),
(np.sqrt(6)/np.sqrt(ne+he)), size=(ne, he))

        uzar = np.random.uniform(-(np.sqrt(6)/np.sqrt(he+he)), (np.sqrt(6)/np.sqrt(he+he)),
size=(he, he))

        biasr = np.zeros((1, he))

        weightsh = np.random.uniform(-(np.sqrt(6)/np.sqrt(ne+he)),
(np.sqrt(6)/np.sqrt(ne+he)), size=(ne, he))
```

```
        uzah = np.random.uniform(-(np.sqrt(6)/np.sqrt(he+he)),
(np.sqrt(6)/np.sqrt(he+he)), size=(he, he))

        biash = np.zeros((1, he))


        params = {"weightsz":weightsz, "uzaz": uzaz, "biasz": biasz, "weightsr":weightsr,
"uzar": uzar, "biasr": biasr, "weightsh":weightsh, "uzah": uzah, "biash": biash}


    mom = dict.fromkeys(params.keys(), 0)

    self.flaypar = params

    self.flaymom = mom


  def modify(self, lrate, momc, gradflay, gradperc):


    flaypar = self.flaypar

    flaymom = self.flaymom

    percpar = self.percpar

    percmom = self.percmom


    for k in self.flaypar:

      flaymom[k] = lrate * gradflay[k] + momc * flaymom[k]

      flaypar[k] = flaypar[k] - (lrate * gradflay[k] + momc * flaymom[k])


    for i in range(self.percbig):


        percmom["weights"][i] = lrate*gradperc["weights"][i] + momc *
percmom["weights"][i]

        percmom["bias"][i] = lrate * gradperc["bias"][i] + momc * percmom["bias"][i]


        percpar["weights"][i] = percpar["weights"][i] - (lrate*gradperc["weights"][i] + momc
* percmom["weights"][i])
```

```
            percpar["bias"][i] = percpar["bias"][i] - (lrate * gradperc["bias"][i] + momc *
percmom["bias"][i])


        self.flaypar = flaypar
        self.flaymom = flaymom
        self.percpar = percpar
        self.percmom = percmom

    def ileriperc(self,a,weights,bias,b):
        return self.activ((a @ weights + bias), b)

    def geriperc(self, weights, o, der, chan):

        dweights = o.T @ chan
        dbias = chan.sum(axis=0, keepdims=True)
        chan = der * (chan @ weights.T)
        return dweights, dbias, chan

    def workout3(self, a, b, lrate, momc, batch, epoch):
        traininglossl, validationlossl, trainingaccuracyl, validationaccuracyl = [], [], [], []

        valbig = int(a.shape[0]/10)
        po = np.random.permutation(a.shape[0])
        vala = a[po][:valbig]
        valb = b[po][:valbig]
        a = a[po][valbig:]
        b = b[po][valbig:]
        it = int(a.shape[0]/batch)
```

```
        for i in range(epoch):
            go = 0
            stop = batch
            po = np.random.permutation(a.shape[0])
            a = a[po]
            b = b[po]

            for j in range(it):
                gues, o, der, h, hder, co = self.ilerigo(a[go:stop])

                chan = gues
                chan[b[go:stop] ==1] = chan[b[go:stop] ==1]-1
                chan = chan/batch

                gradflay, gradperc = self.gerigo(a[go:stop], o, der, chan, h, hder, co)

                self.modify(lrate, momc, gradflay, gradperc)

                go = stop
                stop = stop + batch

            trainingaccuracy, trainingloss = calcul(a,b,self.guess3,self.CE)
            validationaccuracy, validationloss = calcul(vala,valb,self.guess3,self.CE)

            print("Epoch: %d | Training Loss: %.3f, Validation Loss: %.3f, Training Accuracy:
%.3f, Validation Accuracy: %.3f"% (i + 1, trainingloss, validationloss, trainingaccuracy,
validationaccuracy))
            traininglossl.append(trainingloss)
            validationlossl.append(validationloss)
```

```
            trainingaccuracyl.append(trainingaccuracy)
            validationaccuracyl.append(validationaccuracy)



            if i>15:
                convergence = sum(validationlossl[-16:-1]) / len(validationlossl[-16:-1])
                if (convergence - 0.001) < validationloss < (convergence + 0.001):
                    print("\nTraining stopped since validation C-E reached convergence.")
                    return {"traininglossl": traininglossl, "validationlossl": validationlossl,
"trainingaccuracyl": trainingaccuracyl, "validationaccuracyl": validationaccuracyl}
        return {"traininglossl": traininglossl, "validationlossl": validationlossl,
"trainingaccuracyl": trainingaccuracyl, "validationaccuracyl": validationaccuracyl}


    def activ(self, a, A):
        if A == "softmax":
            activ = np.exp(a) / np.sum(np.exp(a), axis=1, keepdims=True)
            deriv = None
            return activ, deriv

        if A == "tanh":
            activ = np.tanh(a)
            deriv = 1 - activ**2
            return activ, deriv

        if A == "relu":
            activ = a * (a>0)
            deriv = 1 * (a>0)
            return activ, deriv
```

```
        if A == "sigmoid":
            activ = np.exp(a)/(1 + np.exp(a))
            deriv = activ * (1-activ)
            return activ, deriv

    def ilerirnn(self, a, flaypar):

        ne, te, de = a.shape

        weightsih = flaypar["weightsih"]
        weightshh = flaypar["weightshh"]
        bias = flaypar["bias"]

        hbefore = np.zeros((ne, self.big[1]))
        h, hder = np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1]))

        for k in range(te):
            h[:, k, :], hder[:, k, :] = self.activ((a[:, k, :] @ weightsih + hbefore @ weightshh +
bias), "tanh")
            hbefore = h[:, k, :]

        return h, hder

    def gerirnn(self, a, h, hder, chan, flaypar):

        ne, te, de = a.shape
        weightshh = flaypar["weightshh"]
        dweightsih, dweightshh, dbias = 0,0,0
```

```
    for k in reversed(range(te)):
        hbefore = ifing1(h, self.big[1], ne, k)
        hbeforeder = ifing2(hder, k)
        dweightsih = dweightsih + a[:, k, :].T @ chan
        dweightshh = dweightshh + hbefore.T @ chan
        dbias = dbias + chan.sum(axis=0, keepdims=True)
        chan = hbeforeder * (chan@weightshh)

    return {"weightsih": dweightsih, "weightshh": dweightshh, "bias": dbias}

def ilerigo(self,a):

    num = self.num
    percpar = self.percpar
    flaypar = self.flaypar
    o, der = [], []
    h, hder, co = 0,0,0

    if num == 1:
        h, hder = self.ilerirnn(a, flaypar)
        o.append(h[:,-1,:])
        der.append(hder[:,-1,:])

    if num == 2:
        h, co = self.ilerilstm(a, flaypar)
        o.append(h)
        der.append(1)

    if num == 3:
```

```
            h, co = self.ilerigru(a,flaypar)
            o.append(h)
            der.append(1)

        for i in range(self.percbig-1):
            activ, deriv = self.ileriperc(o[-1], percpar["weights"][i], percpar["bias"][i], "relu")
            o.append(activ)
            der.append(deriv)

        gues = self.ileriperc(o[-1], percpar["weights"][-1], percpar["bias"][-1], "softmax")[0]

        return gues, o, der, h, hder, co

    def gerigo(self, a, o, der, chan, h, hder, co):

        num = self.num
        percpar = self.percpar
        flaypar = self.flaypar

        gradflay = dict.fromkeys(percpar.keys())
        gradperc = {"weights": [0] * self.percbig, "bias": [0]*self.percbig}

        for i in reversed(range(self.percbig)):
            gradperc["weights"][i], gradperc["bias"][i], chan =
self.geriperc(percpar["weights"][i], o[i], der[i], chan)

        if num == 1:
            gradflay = self.gerirnn(a, h, hder, chan, flaypar)
        if num == 2:
```

```
            gradflay = self.gerilstm(co, flaypar, chan)
        if num == 3:
            gradflay = self.gerigru(a, co, flaypar, chan)

        return gradflay, gradperc

    def ilerilstm(self, a, flaypar):

        ne, te, de = a.shape

        weightsi, biasi = flaypar["weightsi"], flaypar["biasi"]
        weightsf, biasf = flaypar["weightsf"], flaypar["biasf"]
        weightso, biaso = flaypar["weightso"], flaypar["biaso"]
        weightsc, biasc = flaypar["weightsc"], flaypar["biasc"]

        hbefore, cbefore = np.zeros((ne, self.big[1])), np.zeros((ne, self.big[1]))
        zi = np.empty((ne, te, de + self.big[1]))
        hfi = 0

        hii, hci, hoi, tanhci, ci, tanhcdi, hfdi, hidi, hcdi, hodi = np.empty((ne, te, self.big[1])),
np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])),
np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])),
np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1]))

        for k in range(te):
            zi[:, k, :] = np.column_stack((hbefore, a[:, k, :]))

            hfi, hfdi[:, k, :] = self.activ(zi[:, k, :] @ weightsf + biasf, "sigmoid")
            hii[:, k, :], hidi[:, k, :] = self.activ(zi[:, k, :] @ weightsi + biasi, "sigmoid")
            hci[:, k, :], hcdi[:, k, :] = self.activ(zi[:, k, :] @ weightsc + biasc, "tanh")
```

```
            hoi[:, k, :], hodi[:, k, :] = self.activ(zi[:, k, :] @ weightso + biaso, "sigmoid")


            ci[:, k, :] = hfi * cbefore + hii[:, k, :] * hci[:, k, :]
            tanhci[:, k, :], tanhcdi[:, k, :] = self.activ(ci[:, k, :], "tanh")
            hbefore = hoi[:, k, :] * tanhci[:, k, :]
            cbefore = ci[:, k, :]


            co = {"zi": zi, "ci": ci, "tanhci": (tanhci, tanhcdi), "hfdi": hfdi, "hii": (hii, hidi), "hci":
(hci, hcdi), "hoi": (hoi, hodi)}


        return hbefore, co


    def gerilstm(self, co, flaypar, chan):


        weightsf = flaypar["weightsf"]
        weightsi = flaypar["weightsi"]
        weightsc = flaypar["weightsc"]
        weightso = flaypar["weightso"]


        zi = co["zi"]
        ci = co["ci"]
        tanhci, tanhcdi = co["tanhci"]
        hfdi = co["hfdi"]
        hii, hidi = co["hii"]
        hci, hcdi = co["hci"]
        hoi, hodi = co["hoi"]
        te = zi.shape[1]
```

```
        dweightsf, dweightsi, dweightsc, dweightso, dbiasf, dbiasi, dbiasc, dbiaso =
0,0,0,0,0,0,0,0


        for k in reversed(range(te)):
            cbefore = ifing2(ci, k)


            dci = chan * hoi[:, k, :] * tanhcdi[:, k, :]
            dhfi = dci * cbefore * hfdi[:, k, :]
            dhii = dci * hci[:, k, :] * hidi[:, k, :]
            dhci = dci * hii[:, k, :] * hcdi[:, k, :]
            dhoi = chan * tanhci[:, k, :] * hodi[:, k, :]


            dweightsf, dbiasf = lstmproc(dweightsf, dbiasf, zi[:, k, :], dhfi)
            dweightsi, dbiasi = lstmproc(dweightsi, dbiasi, zi[:, k, :], dhii)
            dweightsc, dbiasc = lstmproc(dweightsc, dbiasc, zi[:, k, :], dhci)
            dweightso, dbiaso = lstmproc(dweightso, dbiaso, zi[:, k, :], dhoi)


            df = lstmdu(dhfi, weightsf, self.big[1])
            di = lstmdu(dhii, weightsi, self.big[1])
            dc = lstmdu(dhci, weightsc, self.big[1])
            do = lstmdu(dhoi, weightso, self.big[1])



            chan = (df + di + dc + do)


        return {"weightsf": dweightsf, "biasf": dbiasf, "weightsi": dweightsi, "biasi": dbiasi,
"weightsc": dweightsc, "biasc": dbiasc, "weightso": dweightso, "biaso": dbiaso}


    def ilerigru(self, a, flaypar):
```

```
        weightsz = flaypar["weightsz"]
        weightsr = flaypar["weightsr"]
        weightsh = flaypar["weightsh"]

        uzaz = flaypar["uzaz"]
        uzar = flaypar["uzar"]
        uzah = flaypar["uzah"]

        biasz = flaypar["biasz"]
        biasr = flaypar["biasr"]
        biash = flaypar["biash"]

        ne, te, de = a.shape
        hbefore = np.zeros((ne, self.big[1]))
        zi, zdi, ri, rdi, htider, htiderd, hi = np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1]))

        for k in range(te):
            zi[:, k, :], zdi[:, k, :] = self.activ(a[:, k, :] @ weightsz + hbefore @ uzaz + biasz, "sigmoid")
            ri[:, k, :], rdi[:, k, :] = self.activ(a[:, k, :] @ weightsr + hbefore @ uzar + biasr, "sigmoid")
            htider[:, k, :], htiderd[:, k, :] = self.activ(a[:, k, :] @ weightsh + (ri[:, k, :] * hbefore) @ uzah + biash, "tanh")
            hi[:, k, :] = (1 - zi[:, k, :]) * hbefore + zi[:, k, :] * htider[:, k, :]

            hbefore = hi[:, k, :]

        co = {"zi": (zi, zdi), "ri": (ri, rdi), "htider": (htider, htiderd), "hi": hi}
```

```
        return hbefore, co


    def gerigru(self, a, co, flaypar, chan):

        uzaz = flaypar["uzaz"]
        uzar = flaypar["uzar"]
        uzah = flaypar["uzah"]

        zi, zdi = co["zi"]
        ri, rdi = co["ri"]
        htider, htiderd = co["htider"]
        hi = co["hi"]

        ne, te, de = a.shape
        dweightsz, dweightsr, dweightsh, duzaz, duzar, duzah, dbiasz, dbiasr, dbiash =
0,0,0,0,0,0,0,0,0

        for k in reversed(range(te)):
            hbefore = ifing1(hi, self.big[1], ne, k)

            dzi = chan * (htider[:, k, :] - hbefore) * zdi[:, k, :]
            dhtider = chan * zi[:, k, :] * htiderd[:, k, :]
            dri = (dhtider @ uzah.T) * hbefore * rdi[:, k, :]

            dweightsz, duzaz, dbiasz = gruproc(a[:, k, :], dzi, hbefore, dweightsz, duzaz, dbiasz)
            dweightsr, duzar, dbiasr = gruproc(a[:, k, :], dri, hbefore, dweightsr, duzar, dbiasr)
            dweightsh, duzah, dbiash = gruproc(a[:, k, :], dhtider, hbefore, dweightsh, duzah,
dbiash)
```

```
            chan = (chan * (1 - zi[:, k, :])) + (dzi @ uzaz.T) + ((dhtider @ uzah.T) * (ri[:, k, :] +
hbefore * (rdi[:, k, :] @ uzar.T)))


        return {"weightsz": dweightsz, "uzaz": duzaz, "biasz": dbiasz, "weightsr": dweightsr,
"uzar": duzar, "biasr": dbiasr, "weightsh": dweightsh, "uzah": duzah, "biash": dbiash}


    def guess3(self, a, b=None, accur=True, conf=False):


        guessino = self.ilerigo(a)[0]


        if not accur:
            return guessino


        guessino = guessino.argmax(axis=1)
        b = b.argmax(axis=1)


        if not conf:
            return (guessino == b).mean() * 100


        cla = np.zeros((len(np.unique(b)),len(np.unique(b))))


        for k in range(len(b)):
            cla[b[k]][guessino[k]] = cla[b[k]][guessino[k]] + 1


        return cla


    def CE(self, d, y):
        return np.sum(np.log(y) * -d) / d.shape[0]
```

```
filename3 = "assign3_data3.h5"
f3 = h5py.File(filename3, 'r')

trainx = np.array(f3['trX'])
trainy = np.array(f3['trY'])
testx = np.array(f3['tstX'])
testy = np.array(f3['tstY'])



"""Part a"""

print("Question 3 Part a")
print("Recurrent Layer\n")
epoch3rnn = 50
lrate3rnn = 0.01
batch3rnn = 32
momc3rnn = 0.85
bigrnn = [trainx.shape[2], 128, 32, 16, 6]

net3rnn, traininglosslrnn, validationlosslrnn, trainingaccuracylrnn, validationaccuracylrnn, testaccuracyrnn = getting(Network3, bigrnn, trainx, trainy, lrate3rnn, momc3rnn, batch3rnn, epoch3rnn, testx, testy, 1)
print("\nTest Accuracy: ", testaccuracyrnn, "\n\n")

trainingconfrnn, testingconfrnn = guessing(net3rnn, trainx, trainy, testx, testy)

"""Part b"""
print("\nQuestion 3 Part b")
```

```
print("LSTM Layer\n")


epoch3lstm = 50

lrate3lstm = 0.01

batch3lstm = 32

momc3lstm = 0.85

biglstm = [trainx.shape[2], 128, 32, 16, 6]


net3lstm, traininglossllstm, validationlossllstm, trainingaccuracyllstm,
validationaccuracyllstm, testaccuracylstm = getting(Network3, biglstm, trainx, trainy,
lrate3lstm, momc3lstm, batch3lstm, epoch3lstm, testx, testy, 2)

print("\nTest Accuracy: ", testaccuracylstm, "\n\n")


trainingconflstm, testingconflstm = guessing(net3lstm, trainx, trainy, testx, testy)


"""Part c"""

print("\nQuestion 3 Part c")

print("GRU Layer\n")


epoch3gru = 50

lrate3gru = 0.01

batch3gru = 32

momc3gru = 0.85

biggru = [trainx.shape[2], 128, 32, 16, 6]


net3gru, traininglosslgru, validationlosslgru, trainingaccuracylgru, validationaccuracylgru,
testaccuracygru= getting(Network3, biggru, trainx, trainy, lrate3gru, momc3gru, batch3gru,
epoch3gru, testx, testy, 3)

print("\nTest Accuracy: ", testaccuracygru, "\n\n")
```

```
    trainingconfgru, testingconfgru = guessing(net3gru, trainx, trainy, testx, testy)


    """ Plots"""
    print('To continue executing, after observation please close the plots')
    print("\nQuestion 3, Part a Plots")
    graphn(trainingaccuracylrnn, validationaccuracylrnn, testaccuracyrnn, traininglosslrnn,
"RNN")
    plt.title("Training Cross Entropy Loss")
    plt.ylabel("Loss")
    plt.show()


    graphn(trainingaccuracylrnn, validationaccuracylrnn, testaccuracyrnn, validationlosslrnn,
"RNN")
    plt.title("Validation Cross Entropy Loss")
    plt.ylabel("Loss")
    plt.show()


    graphn(trainingaccuracylrnn, validationaccuracylrnn, testaccuracyrnn,
trainingaccuracylrnn, "RNN")
    plt.title("Training Accuracy")
    plt.ylabel("Accuracy")
    plt.show()


    graphn(trainingaccuracylrnn, validationaccuracylrnn, testaccuracyrnn,
validationaccuracylrnn, "RNN")
    plt.title("Validation Accuracy")
    plt.ylabel("Accuracy")
    plt.show()


    graphm(trainingconfrnn, testingconfrnn)
```

```
    plt.show()



    print("\nQuestion 3, Part b Plots")
    graphn(trainingaccuracyllstm, validationaccuracyllstm, testaccuracylstm, trainingloss llstm, "LSTM")
    plt.title("Training Cross Entropy Loss")
    plt.ylabel("Loss")
    plt.show()

    graphn(trainingaccuracyllstm, validationaccuracyllstm, testaccuracylstm, validationloss llstm, "LSTM")
    plt.title("Validation Cross Entropy Loss")
    plt.ylabel("Loss")
    plt.show()

    graphn(trainingaccuracyllstm, validationaccuracyllstm, testaccuracylstm, trainingaccuracyllstm, "LSTM")
    plt.title("Training Accuracy")
    plt.ylabel("Accuracy")
    plt.show()

    graphn(trainingaccuracyllstm, validationaccuracyllstm, testaccuracylstm, validationaccuracyllstm, "LSTM")
    plt.title("Validation Accuracy")
    plt.ylabel("Accuracy")
    plt.show()

    graphm(trainingconflstm, testingconflstm)
    plt.show()
```

```
    print("\nQuestion 3, Part c Plots")

    graphn(trainingaccuracylgru, validationaccuracylgru, testaccuracygru, trainingloss lgru, "GRU")

    plt.title("Training Cross Entropy Loss")

    plt.ylabel("Loss")

    plt.show()


    graphn(trainingaccuracylgru, validationaccuracylgru, testaccuracygru, validationloss lgru, "GRU")

    plt.title("Validation Cross Entropy Loss")

    plt.ylabel("Loss")

    plt.show()


    graphn(trainingaccuracylgru, validationaccuracylgru, testaccuracygru, trainingaccuracylgru, "GRU")

    plt.title("Training Accuracy")

    plt.ylabel("Accuracy")

    plt.show()


    graphn(trainingaccuracylgru, validationaccuracylgru, testaccuracygru, validationaccuracylgru, "GRU")

    plt.title("Validation Accuracy")

    plt.ylabel("Accuracy")

    plt.show()


    graphm(trainingconfgru, testingconfgru)

    plt.show()
```

```
question = sys.argv[1]


def ayberk_yarkin_yildiz_21803386_hw3(question):

   if question == '1' :
      q1()

   elif question == '3' :
      q3()

ayberk_yarkin_yildiz_21803386_hw3(question)
```

Question 2

Convolutional Networks

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

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

In [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_grad
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))))
```

In [2]:
```
# Load the (preprocessed) CIFAR10 data.

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

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

Convolution: Naive forward pass

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

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

You can test your implementation by running the following:

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

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

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

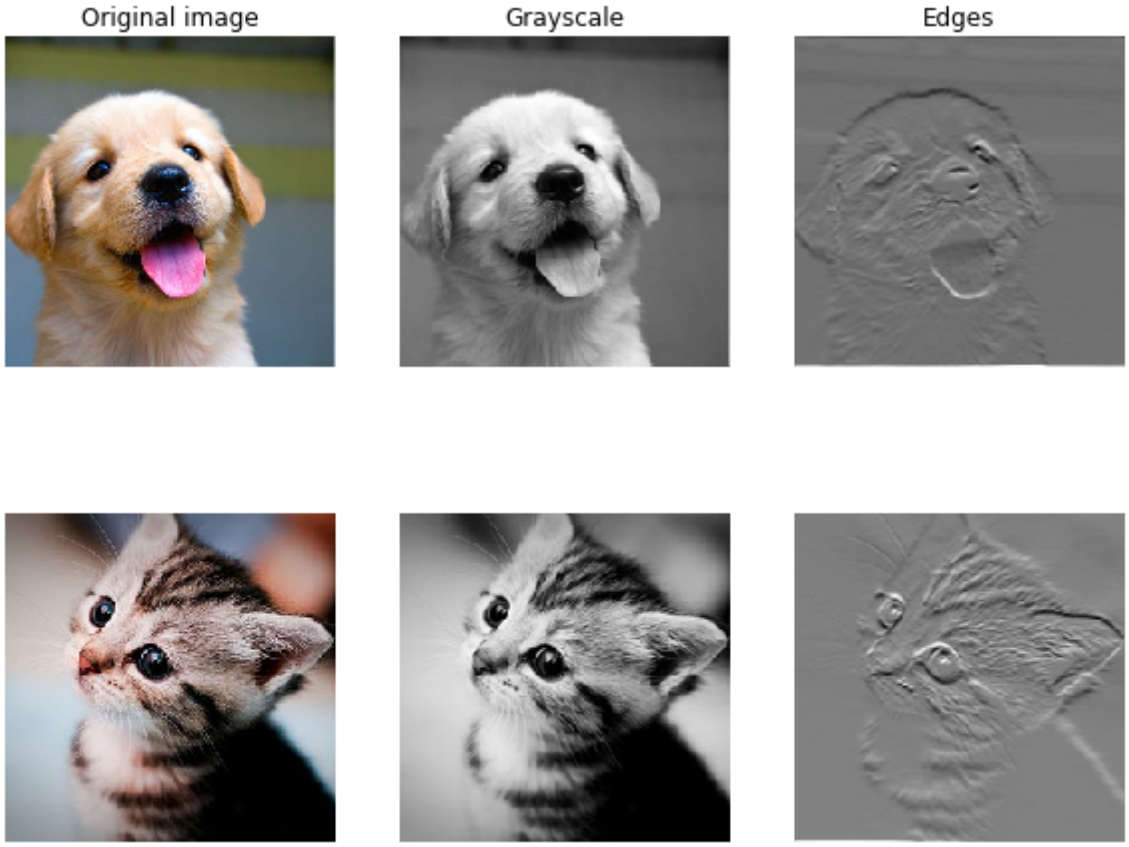
```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

Aside: Image processing via convolutions

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

In [4]:
```
from matplotlib.pyplot import imread
from PIL import Image

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, :, :, :] = np.array(Image.fromarray(puppy).resize((img_size, img_size))).transp
x[1, :, :, :] = np.array(Image.fromarray(kitten_cropped).resize((img_size, img_size)

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

# The first filter converts the image to grayscale.
```

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

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

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

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

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

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



Convolution: Naive backward pass

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

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

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

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b, conv_pa
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_pa
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_pa

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

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

```
Testing conv_backward_naive function
dx error:  1.159803161159293e-08
```

```
dw error:  2.2471264748452487e-10
db error:  3.37264006649648e-11
```

Max-Pooling: Naive forward

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

Check your implementation by running the following:

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

out, _ = max_pool_forward_naive(x, pool_param)

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

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

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

Max-Pooling: Naive backward

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

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

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

dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x, pool_para

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

# Your error should be on the order of e-12
```

```
print('Testing max_pool_backward_naive function:')
print('dx error: ', rel_error(dx, dx_num))
```

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

Fast layers

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

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

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

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

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

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

In [8]:

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

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

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

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

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

```
Testing conv_forward_fast:
Naive: 10.164003s
Fast: 0.017996s
Speedup: 564.790040x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 14.907001s
Fast: 0.014001s
Speedup: 1064.699770x
dx difference:  1.949764775345631e-11
dw difference:  4.4985195578905695e-13
db difference:  0.0
```

In [9]:

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

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

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

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

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

```
Testing pool_forward_fast:
Naive: 0.301002s
fast: 0.015996s
speedup: 18.817355x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.907001s
fast: 0.025002s
speedup: 36.277144x
dx difference:  0.0
```

Convolutional "sandwich" layers

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

In [10]:
```
from cs231n.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
np.random.seed(231)
x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

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

# 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:  6.514336569263308e-09
dw error:  1.490843753539445e-08
db error:  2.037390356217257e-09
```

In [11]:
```
from cs231n.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

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

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b, conv_par
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, conv_par
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, conv_par

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

```
Testing conv_relu:
dx error:  3.5600610115232832e-09
dw error:  2.2497700915729298e-10
db error:  1.3087619975802167e-10
```

Three-layer ConvNet

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

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

Sanity check loss

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

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

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

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

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

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

Gradient check

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

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

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

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

Overfit small data

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

In [14]:

```
np.random.seed(231)

num_train = 100
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

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

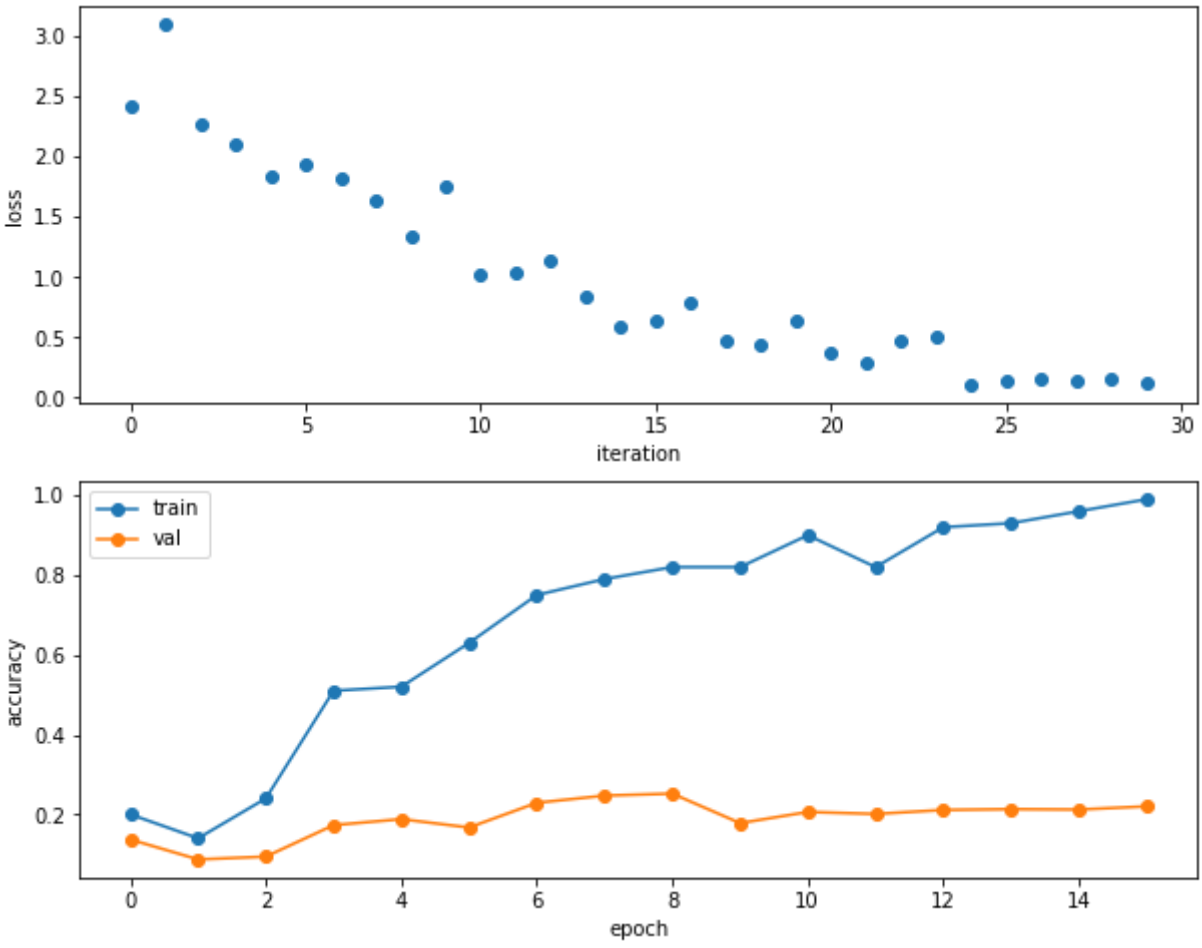
```
(Iteration 1 / 30) loss: 2.414060
(Epoch 0 / 15) train acc: 0.200000; val_acc: 0.137000
(Iteration 2 / 30) loss: 3.102925
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.270330
(Iteration 4 / 30) loss: 2.096705
(Epoch 2 / 15) train acc: 0.240000; val_acc: 0.094000
(Iteration 5 / 30) loss: 1.838880
(Iteration 6 / 30) loss: 1.934188
(Epoch 3 / 15) train acc: 0.510000; val_acc: 0.173000
(Iteration 7 / 30) loss: 1.827912
(Iteration 8 / 30) loss: 1.639574
(Epoch 4 / 15) train acc: 0.520000; val_acc: 0.188000
(Iteration 9 / 30) loss: 1.330082
(Iteration 10 / 30) loss: 1.756115
(Epoch 5 / 15) train acc: 0.630000; val_acc: 0.167000
(Iteration 11 / 30) loss: 1.024162
(Iteration 12 / 30) loss: 1.041826
(Epoch 6 / 15) train acc: 0.750000; val_acc: 0.229000
(Iteration 13 / 30) loss: 1.142777
(Iteration 14 / 30) loss: 0.835706
(Epoch 7 / 15) train acc: 0.790000; val_acc: 0.247000
(Iteration 15 / 30) loss: 0.587786
(Iteration 16 / 30) loss: 0.645509
(Epoch 8 / 15) train acc: 0.820000; val_acc: 0.252000
(Iteration 17 / 30) loss: 0.786844
(Iteration 18 / 30) loss: 0.467054
(Epoch 9 / 15) train acc: 0.820000; val_acc: 0.178000
(Iteration 19 / 30) loss: 0.429880
(Iteration 20 / 30) loss: 0.635498
```

```
(Epoch 10 / 15) train acc: 0.900000; val_acc: 0.206000
(Iteration 21 / 30) loss: 0.365807
(Iteration 22 / 30) loss: 0.284220
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.201000
(Iteration 23 / 30) loss: 0.469343
(Iteration 24 / 30) loss: 0.509369
(Epoch 12 / 15) train acc: 0.920000; val_acc: 0.211000
(Iteration 25 / 30) loss: 0.111638
(Iteration 26 / 30) loss: 0.145388
(Epoch 13 / 15) train acc: 0.930000; val_acc: 0.213000
(Iteration 27 / 30) loss: 0.155575
(Iteration 28 / 30) loss: 0.143398
(Epoch 14 / 15) train acc: 0.960000; val_acc: 0.212000
(Iteration 29 / 30) loss: 0.158160
(Iteration 30 / 30) loss: 0.118934
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.220000
```

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

In [15]:

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

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



Train the net

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

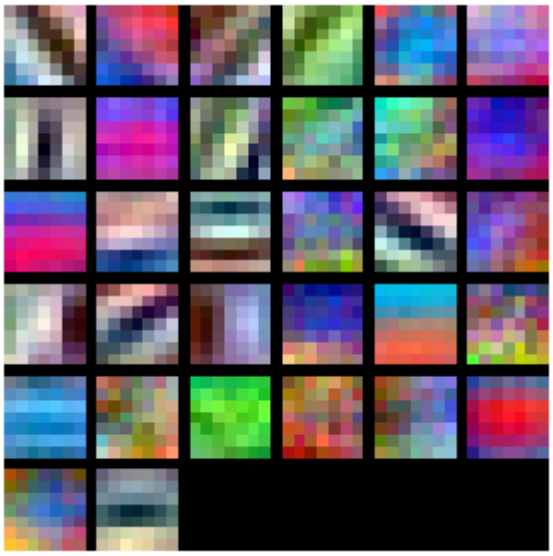
In [16]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

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

```
(Iteration 1 / 980) loss: 2.304740
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.098229
(Iteration 41 / 980) loss: 1.949788
(Iteration 61 / 980) loss: 1.888398
(Iteration 81 / 980) loss: 1.877093
(Iteration 101 / 980) loss: 1.851877
(Iteration 121 / 980) loss: 1.859353
(Iteration 141 / 980) loss: 1.800181
(Iteration 161 / 980) loss: 2.143292
(Iteration 181 / 980) loss: 1.830573
(Iteration 201 / 980) loss: 2.037280
(Iteration 221 / 980) loss: 2.020304
(Iteration 241 / 980) loss: 1.823728
(Iteration 261 / 980) loss: 1.692679
(Iteration 281 / 980) loss: 1.882594
(Iteration 301 / 980) loss: 1.798261
(Iteration 321 / 980) loss: 1.851960
(Iteration 341 / 980) loss: 1.716323
(Iteration 361 / 980) loss: 1.897655
(Iteration 381 / 980) loss: 1.319744
(Iteration 401 / 980) loss: 1.738790
(Iteration 421 / 980) loss: 1.488866
(Iteration 441 / 980) loss: 1.718409
(Iteration 461 / 980) loss: 1.744440
(Iteration 481 / 980) loss: 1.605460
(Iteration 501 / 980) loss: 1.494847
(Iteration 521 / 980) loss: 1.835179
(Iteration 541 / 980) loss: 1.483923
(Iteration 561 / 980) loss: 1.676871
(Iteration 581 / 980) loss: 1.438325
(Iteration 601 / 980) loss: 1.443469
(Iteration 621 / 980) loss: 1.529369
(Iteration 641 / 980) loss: 1.763475
(Iteration 661 / 980) loss: 1.790329
(Iteration 681 / 980) loss: 1.693343
(Iteration 701 / 980) loss: 1.637078
(Iteration 721 / 980) loss: 1.644564
(Iteration 741 / 980) loss: 1.708919
(Iteration 761 / 980) loss: 1.494252
(Iteration 781 / 980) loss: 1.901751
(Iteration 801 / 980) loss: 1.898991
(Iteration 821 / 980) loss: 1.489988
(Iteration 841 / 980) loss: 1.377615
(Iteration 861 / 980) loss: 1.763751
(Iteration 881 / 980) loss: 1.540284
(Iteration 901 / 980) loss: 1.525582
(Iteration 921 / 980) loss: 1.674166
(Iteration 941 / 980) loss: 1.714316
(Iteration 961 / 980) loss: 1.534668
(Epoch 1 / 1) train acc: 0.504000; val_acc: 0.499000
```

Visualize Filters

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

In [17]:
```
from cs231n.vis_utils import visualize_grid

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



Spatial Batch Normalization

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

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

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

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

Spatial batch normalization: forward

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

In [18]:
```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

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

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

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

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

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

In [19]:
```
np.random.seed(231)
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

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

```
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)

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

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

Spatial batch normalization: backward

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

In [20]:

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

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

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

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

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

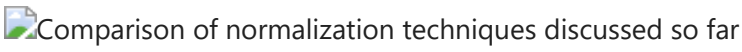
Group Normalization

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

> With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction, and re-centering and rescaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden

> units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

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


Comparison of normalization techniques discussed so far

Visual comparison of the normalization techniques discussed so far (image edited from [5])
Even though an assumption of equal contribution is still being made within each group, the authors hypothesize that this is not as problematic, as innate grouping arises within features for visual recognition. One example they use to illustrate this is that many high-performance handcrafted features in traditional Computer Vision have terms that are explicitly grouped together. Take for example Histogram of Oriented Gradients [6]-- after computing histograms per spatially local block, each per-block histogram is normalized before being concatenated together to form the final feature vector.

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

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

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

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

Group normalization: forward

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

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

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

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

out, _ = spatial_groupnorm_forward(x, gamma, beta, G, bn_param)
```

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

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

Spatial group normalization: backward

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

In [22]:

```
np.random.seed(231)
N, C, H, W = 2, 6, 4, 5
G = 2
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

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

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

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

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

What's this PyTorch business?

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

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

What is PyTorch?

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

Why?

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

PyTorch versions

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

How will I learn PyTorch?

Justin Johnson has made an excellent tutorial for PyTorch.

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

Table of Contents

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

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

Here is a table of comparison:

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

Part I. Preparation

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

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

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

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

import numpy as np
```

In [2]:
```
NUM_TRAIN = 49000

# The torchvision.transforms package provides tools for preprocessing data
# and for performing data augmentation; here we set up a transform to
# preprocess the data by subtracting the mean RGB value and dividing by the
# standard deviation of each RGB value; we've hardcoded the mean and std.
```

```
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
            ])

# We set up a Dataset object for each split (train / val / test); Datasets load
# training examples one at a time, so we wrap each Dataset in a DataLoader which
# iterates through the Dataset and forms minibatches. We divide the CIFAR-10
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar10_train = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                             transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

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

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

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

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

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

In [3]:

```
USE_GPU = True

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

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

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

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

```
using device: cuda
```

Part II. Barebones PyTorch

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

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

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

PyTorch Tensors: Flatten Function

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

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

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

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

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

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

test_flatten()
```

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

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

Barebones PyTorch: Two-Layer Network

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

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

In [5]:

```
import torch.nn.functional as F  # useful stateless functions

def two_layer_fc(x, params):
    """
    A fully-connected neural networks; the architecture is:
    NN is fully connected -> ReLU -> fully connected layer.
    Note that this function only defines the forward pass;
    PyTorch will take care of the backward pass for us.

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

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

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

    w1, w2 = params

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


def  two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature dimension 5
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
```

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

two_layer_fc_test()
```

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

Barebones PyTorch: Three-Layer ConvNet

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

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

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

In [6]:

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

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

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

    conv1 = F.conv2d(x, weight=conv_w1, bias=conv_b1, padding=2)
    relu1 = F.relu(conv1)
    conv2 = F.conv2d(relu1, weight=conv_w2, bias=conv_b2, padding=1)
```

```
    relu2 = F.relu(conv2)
    relu2_flat = flatten(relu2)
    scores = relu2_flat.mm(fc_w) + fc_b

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

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

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

In [7]:

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

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

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

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
    print(scores.size())  # you should see [64, 10]
three_layer_convnet_test()
```

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

Barebones PyTorch: Initialization

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

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

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

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

In [8]:

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

```
    return w

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

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

Out[8]:
```
tensor([[ 0.4123, -0.3226,  0.1429,  0.3334,  1.9862],
        [ 0.3655,  1.2438, -1.0655,  0.6376,  1.3459],
        [-0.1194, -0.6314, -0.5256,  0.3198,  0.5616]], device='cuda:0',
       requires_grad=True)
```

Barebones PyTorch: Check Accuracy

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

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

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

    Inputs:
    - loader: A DataLoader for the data split we want to check
    - model_fn: A function that performs the forward pass of the model,
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the model

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

BareBones PyTorch: Training Loop

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

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

In [10]:

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

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

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

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

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

        # Update parameters. We don't want to backpropagate through the
        # parameter updates, so we scope the updates under a torch.no_grad()
        # context manager to prevent a computational graph from being built.
        with torch.no_grad():
            for w in params:
                w -= learning_rate * w.grad

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

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

BareBones PyTorch: Train a Two-Layer Network

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

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

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

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

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

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

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

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

```
Iteration 0, loss = 3.2817
Checking accuracy on the val set
Got 125 / 1000 correct (12.50%)

Iteration 100, loss = 2.2560
Checking accuracy on the val set
Got 320 / 1000 correct (32.00%)

Iteration 200, loss = 1.9334
Checking accuracy on the val set
Got 360 / 1000 correct (36.00%)

Iteration 300, loss = 1.7549
Checking accuracy on the val set
Got 334 / 1000 correct (33.40%)

Iteration 400, loss = 1.9019
Checking accuracy on the val set
Got 402 / 1000 correct (40.20%)

Iteration 500, loss = 1.8647
Checking accuracy on the val set
Got 425 / 1000 correct (42.50%)

Iteration 600, loss = 2.0816
Checking accuracy on the val set
Got 409 / 1000 correct (40.90%)

Iteration 700, loss = 1.9150
Checking accuracy on the val set
Got 416 / 1000 correct (41.60%)
```

BareBones PyTorch: Training a ConvNet

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

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

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

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

In [12]:
```
learning_rate = 3e-3

channel_1 = 32
channel_2 = 16

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

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

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

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

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

```
Iteration 0, loss = 3.0587
Checking accuracy on the val set
Got 96 / 1000 correct (9.60%)

Iteration 100, loss = 1.8284
Checking accuracy on the val set
Got 348 / 1000 correct (34.80%)

Iteration 200, loss = 1.7841
Checking accuracy on the val set
Got 389 / 1000 correct (38.90%)

Iteration 300, loss = 1.7071
Checking accuracy on the val set
Got 425 / 1000 correct (42.50%)

Iteration 400, loss = 1.6384
Checking accuracy on the val set
Got 415 / 1000 correct (41.50%)

Iteration 500, loss = 1.5265
Checking accuracy on the val set
Got 438 / 1000 correct (43.80%)

Iteration 600, loss = 1.5509
Checking accuracy on the val set
Got 465 / 1000 correct (46.50%)

Iteration 700, loss = 1.8430
Checking accuracy on the val set
Got 476 / 1000 correct (47.60%)
```

Part III. PyTorch Module API

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

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

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.
2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!
3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__`as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

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

Module API: Two-Layer Network

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

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

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

def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64, feature dim
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
```

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

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

Module API: Three-Layer ConvNet

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

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

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

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

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

In [14]:

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

        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2, bias
        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=
        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 [
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8, num_classes=1
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_ThreeLayerConvNet()
```

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

Module API: Check Accuracy

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

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

In [15]:

```
def check_accuracy_part34(loader, model):
    if loader.dataset.train:
        print('Checking accuracy on validation set')
    else:
        print('Checking accuracy on test set')
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 * acc))
```

Module API: Training Loop

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

In [16]:

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

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

    Returns: Nothing, but prints model accuracies during training.
```

```
    """
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

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

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

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

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

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

Module API: Train a Two-Layer Network

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

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

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

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

In [17]:

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.9216
Checking accuracy on validation set
Got 147 / 1000 correct (14.70)

Iteration 100, loss = 2.6885
Checking accuracy on validation set
Got 332 / 1000 correct (33.20)

Iteration 200, loss = 1.9235
Checking accuracy on validation set
Got 377 / 1000 correct (37.70)
```

```
Iteration 300, loss = 1.8133
Checking accuracy on validation set
Got 425 / 1000 correct (42.50)

Iteration 400, loss = 1.3607
Checking accuracy on validation set
Got 429 / 1000 correct (42.90)

Iteration 500, loss = 2.1195
Checking accuracy on validation set
Got 388 / 1000 correct (38.80)

Iteration 600, loss = 1.5725
Checking accuracy on validation set
Got 434 / 1000 correct (43.40)

Iteration 700, loss = 1.5672
Checking accuracy on validation set
Got 455 / 1000 correct (45.50)
```

Module API: Train a Three-Layer ConvNet

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

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

In [18]:

```
learning_rate = 3e-3
channel_1 = 32
channel_2 = 16

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

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

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 4.1012
Checking accuracy on validation set
Got 101 / 1000 correct (10.10)

Iteration 100, loss = 1.7691
Checking accuracy on validation set
Got 347 / 1000 correct (34.70)

Iteration 200, loss = 1.8306
Checking accuracy on validation set
Got 397 / 1000 correct (39.70)

Iteration 300, loss = 1.8571
Checking accuracy on validation set
Got 409 / 1000 correct (40.90)
```

```
Iteration 400, loss = 1.5962
Checking accuracy on validation set
Got 433 / 1000 correct (43.30)

Iteration 500, loss = 1.5841
Checking accuracy on validation set
Got 444 / 1000 correct (44.40)

Iteration 600, loss = 1.6299
Checking accuracy on validation set
Got 458 / 1000 correct (45.80)

Iteration 700, loss = 1.3607
Checking accuracy on validation set
Got 476 / 1000 correct (47.60)
```

Part IV. PyTorch Sequential API

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

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

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

Sequential API: Two-Layer Network

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

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

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

hidden_layer_size = 4000
learning_rate = 1e-2

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

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

```
train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3569
Checking accuracy on validation set
Got 159 / 1000 correct (15.90)

Iteration 100, loss = 1.9769
Checking accuracy on validation set
Got 400 / 1000 correct (40.00)

Iteration 200, loss = 1.5860
Checking accuracy on validation set
Got 400 / 1000 correct (40.00)

Iteration 300, loss = 1.7836
Checking accuracy on validation set
Got 404 / 1000 correct (40.40)

Iteration 400, loss = 1.6260
Checking accuracy on validation set
Got 448 / 1000 correct (44.80)

Iteration 500, loss = 1.7615
Checking accuracy on validation set
Got 445 / 1000 correct (44.50)

Iteration 600, loss = 1.7117
Checking accuracy on validation set
Got 446 / 1000 correct (44.60)

Iteration 700, loss = 1.8356
Checking accuracy on validation set
Got 404 / 1000 correct (40.40)
```

Sequential API: Three-Layer ConvNet

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

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

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

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

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

In [20]:

```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
```

```
optimizer = None

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

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

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

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

model.apply(init_weights)

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3153
Checking accuracy on validation set
Got 101 / 1000 correct (10.10)

Iteration 100, loss = 1.5728
Checking accuracy on validation set
Got 449 / 1000 correct (44.90)

Iteration 200, loss = 1.4987
Checking accuracy on validation set
Got 501 / 1000 correct (50.10)

Iteration 300, loss = 1.2147
Checking accuracy on validation set
Got 517 / 1000 correct (51.70)

Iteration 400, loss = 1.1304
Checking accuracy on validation set
Got 501 / 1000 correct (50.10)

Iteration 500, loss = 1.1010
Checking accuracy on validation set
Got 544 / 1000 correct (54.40)

Iteration 600, loss = 1.3214
Checking accuracy on validation set
Got 581 / 1000 correct (58.10)

Iteration 700, loss = 1.4248
Checking accuracy on validation set
```

```
Got 572 / 1000 correct (57.20)
```

Part V. CIFAR-10 open-ended challenge

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

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

Describe what you did at the end of this notebook.

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

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

Things you might try:

- **Filter size**: Above we used 5x5; would smaller filters be more efficient?
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?
- **Batch normalization**: Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster?
- **Network architecture**: The network above has two layers of trainable parameters. Can you do better with a deep network? Good architectures to try include:
  - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling**: Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1 image picture (1, 1 , Filter#), which is then reshaped into a (Filter#) vector. This is used in Google's Inception Network (See Table 1 for their architecture).
- **Regularization**: Add l2 weight regularization, or perhaps use Dropout.

Tips for training

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

- If the parameters are working well, you should see improvement within a few hundred iterations

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

Going above and beyond

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

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

Have fun and happy training!

In [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)
)

layer4 = nn.Sequential(
    nn.Dropout(0.3),
    nn.Linear(64*4*4,  10),
    nn.ReLU(),
    nn.Linear(10,10)
)

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

learning_rate = 1.2e-3

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

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

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

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

```
Iteration 0, loss = 2.3417
Checking accuracy on validation set
Got 87 / 1000 correct (8.70)

Iteration 0, loss = 1.1359
Checking accuracy on validation set
Got 546 / 1000 correct (54.60)

Iteration 0, loss = 0.8376
Checking accuracy on validation set
Got 638 / 1000 correct (63.80)

Iteration 0, loss = 0.9256
Checking accuracy on validation set
Got 665 / 1000 correct (66.50)

Iteration 0, loss = 0.9029
Checking accuracy on validation set
Got 669 / 1000 correct (66.90)

Iteration 0, loss = 1.0565
Checking accuracy on validation set
Got 686 / 1000 correct (68.60)

Iteration 0, loss = 0.7262
Checking accuracy on validation set
Got 677 / 1000 correct (67.70)

Iteration 0, loss = 0.7773
Checking accuracy on validation set
```

```
Got 699 / 1000 correct (69.90)

Iteration 0, loss = 0.8339
Checking accuracy on validation set
Got 706 / 1000 correct (70.60)

Iteration 0, loss = 0.9034
Checking accuracy on validation set
Got 738 / 1000 correct (73.80)
```

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

I firstly understood the network and did some tests about the network, finally I managed to increase the test accuracy to 73.08%. For this increase, I increased my layer amount, that can be seen above as "layer 4". I increased my learning rate. After the convolutional layers, I added a dropout layer to avoid the overfitting, which also gave me the chance to increase the learning rate since the chance of overfitting is decreased. Lastly, I put a ReLU layer to relatively increase the complex learning of the network.

Test set -- run this only once

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

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

```
Checking accuracy on test set
Got 7308 / 10000 correct (73.08)
```

In [ ]:
```

```