Ayhan Okuyan
21601531

EEE443/543 Neural Networks - Assignment 3

Table of Contents

Ayhan Okuyan
21601531

Question 1

This question asks us to implement an autoencoder network with a single hidden layer in order to extract features of an image cluster in an unsupervised manner. In order to implement, we are given the cost function to be minimized which is as follows.

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

Here, the first term represents the Mean Squared Error (MSE) loss, the second term represents an L2 (Tikhonov) regularization in the hidden and output layers regulated with the term lambda and the third term represents the KL Divergence applied to the mean activations of hidden layer regulated with a sparsity term. KL Divergence sparsity is controlled with the term rho.

Part A

This part wants us to implement the preprocessing on the data. The first thing that we do is to convert the RGB images to grayscale using the luminosity model given below.

$$Y = 0.2126 * R + 0.7152 * G + 0.0722 * B$$

After that, we have subtracted the mean of each image from that image and clipped the images according to the [-3*std, 3*std] range where std represents the standard deviation of the image data. Then, in order to prevent the saturation of the data since we are using sigmoid activations in the layers, we have linearly mapped the data to the [0.1,0.9] range. Given the mean shifted grayscale data, the algorithm is as follows.

$$c(X) = \begin{cases} 3 * std\,, if\ x > 3 * std \\ -3 * std\,, if\ x > 3 * std\ , \\ x\ else \end{cases} \qquad \forall x \in X$$

Then we have moved this scale to [0.1,0.9] as follows.

$$s(X, min, max) = \tilde{X} + \left(min - min(\tilde{X})\right), where\ \tilde{X} = X\frac{max - min}{max(X) - min(X)}$$

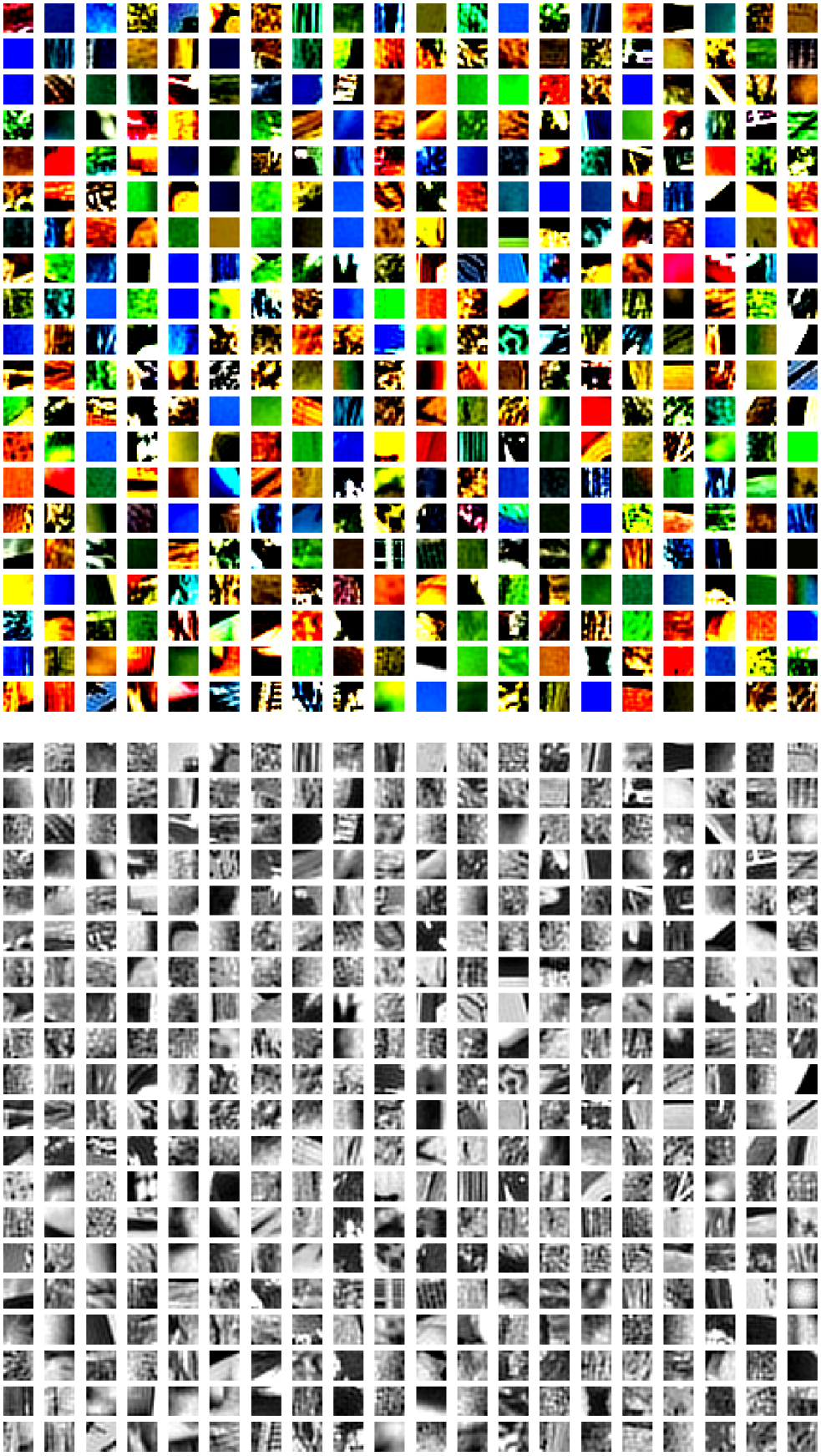
Hence the final data corresponds to,

$$X_f = s(c(X), 0.1, 0.9)$$

The code is as given below.

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

mean = np.mean(data_gs, axis=(1,2))
for i in range(len(mean)):
    data_gs[i,:,:] -= mean[i]

std = np.std(data_gs)
data_nor = np.minimum(data_gs, 3*std)
data_nor = np.maximum(data_nor, -3*std)

minScale = 0.10
maxScale = 0.90
data_nor = data_nor*(maxScale-minScale)/(2*np.max(data_nor))
data_nor += (minScale - np.min(data_nor))
```

After preprocessing, the question asks us to visualize 200 images and their normalized versions. The subplots are given below.



Figures 1-2 – The RGB and Normalized Versions of 200 Randomly Selected Images

Here, we observe that the normalized images resemble the original patches, however, there are some images which look like straight color gradients and their normalized image don't resemble the original image. This is because we are clipping the data for the standard deviation of the entire data. However, the outputs usually resemble the inputs.

Part B

This part asks us to implement the cost function for the autoencoder that would take in the parameters, the data and weights and would return the loss and the gradients of loss according to the weights. However, then the question mentions that we need to optimize the weights using a solver. Solver concept is valid for MATLAB implementations however, we have implemented this assignment using Python and in Python, there is no equivalent of a solver that would minimize the loss according to the loss and derivatives. Hence, in order to observe if the aeCost function is giving valid results, we have followed the given procedure.

1. Implement aeCost function.
2. Implement Full-Batch Gradient Descent Algorithm using aeCost.
3. Implement the equivalent network on Keras.
4. Run both networks on with the same learning rate and epoch.
5. Compare the results.

Hence, we have first implemented the aeCost function. The function asks us to implement the loss described in the beginning of the question and the derivatives. In order to implement the function, we have first coded the forward pass of the network as below.

Forward Pass of the Network

```
def forwardPass(W_e, data):
    W1, W2, b1, b2 = W_e
    W1_e = np.append(W1,b1, axis=0)
    W2_e = np.append(W2,b2, axis=0)

    #input -> hidden
    data_ = np.append(data, np.ones((data.shape[0], 1)), axis=1)
    z = sigmoid(np.matmul(data_,W1_e))

    #hidden -> output
    z_ = np.append(z, np.ones((z.shape[0], 1)), axis=1)
    out = sigmoid(np.matmul(z_,W2_e))

    return z, out
```

Here, we apply the forward pass as we have done for the second assignment as,

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

$$Y = \sigma(W_{Eout} * \sigma(W_{Ehid} * \tilde{X}))$$

$$where\ \sigma(x) = \frac{1}{1+e^{-x}}\ \ and\ W_E = \begin{bmatrix} W \\ \beta \end{bmatrix}, \tilde{X} = [X \quad 1]$$

Then, we have found the derivatives according to each of the terms in the loss separately.

Derivative of MSE

The derivative of loss with respect to weights are derived and calculated in this section. We use the delta rule (partial derivatives) to compute the gradients with respect to the weights and biases. Here we address the part of the loss that is concerned with MSE is given as J_1.

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

$$\delta_{hid} = ((W_{out})^T \delta_{out}) \odot \sigma'(V_{hid})$$

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

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

$$\nabla J_{1_{Whid}} = \delta_{hid}(W_{hid})^T$$

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

Derivative of L2 Regularization

The derivatives of the weight decays are calculated directly as follows.

$$J_2 = \frac{\lambda}{2}\left[\sum_{b=1}^{L_{hid}}\sum_{a=1}^{L_{in}}\left(W_{a,b}^{(1)}\right)^2 + \sum_{c=1}^{L_{out}}\sum_{b=1}^{L_{hid}}\left(W_{b,c}^{(2)}\right)^2\right]$$

$$\nabla J_{2_{Wout}} = \lambda W^{(2)}$$

$$\nabla J_{2_{\beta out}} = 0$$

$$\nabla J_{2_{Whid}} = \lambda W^{(1)}$$

$$\nabla J_{2_{\beta hid}} = 0$$

Derivative of KL Divergence

KL Divergence is a in information theory concept that explains the how different a distribution is from another, here, we look at the divergence of the mean activations of each neuron to a sparsity parameter rho. The KL Divergence is calculated as follows.

$$Given\ a{\sim}Ber(a)\ and\ b{\sim}Ber(b), KL(a|b) = \ a\log\left(\frac{a}{b}\right) + (1-a)\log\left(\frac{1-a}{1-b}\right)$$

Hence the KL term in the loss becomes as follows.

$$J_3 = \beta\sum_{b=1}^{L_{hid}} KL(\rho|\hat{\rho}_b) = \beta\sum_{b=1}^{L_{hid}} \rho\log\left(\frac{\rho}{\hat{\rho}_b}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\hat{\rho}_b}\right)\ \ where\ \hat{\rho}_b = \frac{1}{N}\sum_{i=1}^{N}\sigma\left(W_{Ehid} * \tilde{X}\right)$$

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

$$\nabla J_{2_{\beta out}} = 0$$

$$\nabla J_{2_{Whid}} = \beta\left[-\left(\frac{\rho}{\hat{\rho}_b}\right) + \left(\frac{1-\rho}{1-\hat{\rho}_b}\right)\right]$$

$$\nabla J_{2_{\beta hid}} = 0$$

However, when we observe J_3, we see that this is a column vector. Hence, for the implementation we will tile this term according to the batch size, since we are implementing full-batch, batch size will be the number of data instances. We can restate gradient as,

$$\nabla J_{2Whid} = \left[\beta\left[-\left(\frac{\rho}{\hat{\rho}_b}\right)+\left(\frac{1-\rho}{1-\hat{\rho}_b}\right)\right] \quad \dots \quad \beta\left[-\left(\frac{\rho}{\hat{\rho}_b}\right)+\left(\frac{1-\rho}{1-\hat{\rho}_b}\right)\right]\right]_{(NxL_{Hid})}$$

Gradients with Respect to Loss Function

Hence, by looking at the derivations made, we can express the gradients as,

$$\nabla J_{Whid} = \nabla J_{1Whid} + \nabla J_{2Whid} + \nabla J_{3Whid}$$

$$\boldsymbol{\nabla J_{Whid}} = \frac{\mathbf{1}}{\boldsymbol{N}}\left(\boldsymbol{\delta_{hid}}(\boldsymbol{W_{hid}})^{\boldsymbol{T}} +^{\boldsymbol{c}} \boldsymbol{\beta}\left[-\left(\frac{\boldsymbol{\rho}}{\hat{\boldsymbol{\rho}}_{\boldsymbol{b}}}\right)+\left(\frac{\mathbf{1}-\boldsymbol{\rho}}{\mathbf{1}-\hat{\boldsymbol{\rho}}_{\boldsymbol{b}}}\right)\right]\right) + \boldsymbol{\lambda W^{(1)}}$$

$$\nabla J_{Wout} = \nabla J_{1Wout} + \nabla J_{2Wout} + \nabla J_{3Wout}$$

$$\boldsymbol{\nabla J_{Wout}} = \frac{\mathbf{1}}{\boldsymbol{N}}\boldsymbol{\delta_{out}}(\boldsymbol{W_{out}})^{\boldsymbol{T}} + \boldsymbol{\lambda W^{(2)}}$$

$$\nabla J_{\beta hid} = \nabla J_{1\beta hid} + \nabla J_{2\beta hid} + \nabla J_{3\beta hid}$$

$$\boldsymbol{\nabla J_{\beta hid}} = \frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i}} \boldsymbol{\delta_{hid(i,j)}}$$

$$\nabla J_{\beta out} = \nabla J_{1\beta out} + \nabla J_{2\beta out} + \nabla J_{3\beta out}$$

$$\boldsymbol{\nabla J_{\beta out}} = \frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i}} \boldsymbol{\delta_{out(i,j)}}$$

The sign $+^{\boldsymbol{c}}$ is used to denote that the left side of the addition is a matrix but the right side of it is a column vector. Hence, we implement the code as given below according to the derivations that are made

```
def aeCost(W_e, data, params):
    (Lin, Lhid, lmb, beta, rho) = params
    W1, W2, b1, b2 = W_e
    N = data.shape[0]

    z, dataRec = forwardPass(W_e, data)
    z_mean = np.mean(z, axis=0)

    J1 = (1/(2*N))*np.sum(np.power((data - dataRec),2))
    J2 = (lmb/2)*(np.sum(W1**2) + np.sum(W2**2))
    kl1 = rho*np.log(z_mean/rho)
    kl2 = (1-rho)*np.log((1-z_mean)/(1-rho))
    J3 =  beta*np.sum((kl1+kl2))
    J = J1 + J2 - J3

    deltaOut = -(data-dataRec)*derSigmoid(dataRec)

    derKL = np.tile(beta*(-(rho/z_mean.T)+((1-rho)/(1-z_mean.T))), (10240,1)).T
    deltaHid = (np.matmul(W2,deltaOut.T)+ derKL) * derSigmoid(z).T

    gradWout = (1/N)*(np.matmul(deltaOut.T,z).T + lmb*W2)
    gradBout = np.mean(deltaOut, axis=0)

    gradWhid = (1/N)*(np.matmul(data.T,deltaHid.T) + lmb*W1)
    gradBhid = np.mean(deltaHid, axis=1)
    return J, (gradWhid, gradWout, gradBhid, gradBout)
```

Keras Implementation

Then we applied the same loss function using the Keras API with Tensorflow Backend. In order to apply the same functionality, we have implemented the same KL Divergence loss activity regularizer using the Keras Backend. For L2 Regularization, I have used the already implemented L2 kernel regularizer. The implementation is given below.

```
def kl(rho, beta):
    def customKL(out):
        kl1 = rho*K.log(rho/K.mean(out, axis=0))
        kl2 = (1-rho)*K.log((1-rho)/(1-K.mean(out, axis=0)))
        return beta*K.sum(kl1+kl2)
    return customKL

def mse():
    def customMSE(y_true, y_pred):
        return 0.5*K.sum(K.mean(K.square(y_true-y_pred),axis=0))
    return customMSE
```

Then we have used the Sequential Model to implement the Dense layers as follows.

```
model = Sequential()
model.add(Dense(Lhid, input_shape=(data_flat.shape[1],), activation='sigmoid',\
          kernel_regularizer=regularizers.l2(lmb),  \
          activity_regularizer=kl(rho, beta, data_flat.shape[0])))

model.add(Dense(Lout, activation='sigmoid',\
          kernel_regularizer=regularizers.l2(lmb/2)))
```

Creation of Weights and Biases

Then, the question asked us to define weights according to a uniform distribution as follows,

$$x \sim Uniform([-w_0, w_0]) \; where \; w_0 = \sqrt{\frac{6}{L_{pre} + L_{post}}}$$

Here L_{pre} is the number of input connections for the layer and L_{post} represents the number of units inside the layer. Here is the function we define to define the weight matrices.

```
def w_0(Lpre, Lpost):
    return np.sqrt(6/(Lpre+Lpost))

#Lin=Lout written separately just for formality.
def defWeights(Lin, Lhid, Lout):
    W1 = np.random.uniform(-w_0(Lin,Lhid),w_0(Lin,Lhid),(Lin,Lhid))
    b1 = np.random.uniform(-w_0(Lin,Lhid),w_0(Lin,Lhid),(1,Lhid))
    W2 = np.random.uniform(-w_0(Lhid,Lout),w_0(Lhid,Lout),(Lhid,Lout))
    b2 = np.random.uniform(-w_0(Lhid,Lout),w_0(Lhid,Lout),(1,Lout))
    return (W1, W2, b1, b2)
```

Gradient Descent with aeCost

After completing these we have created a full-batch gradient descent algorithm that will update the weights and biases according to the derivatives, hence we have updated the aeCost to take input the learning rate and output the cost and the updated weight matrices, called as "update". Then we have written the following "sgd" function to set the epochs and update iteratively.

```
def sgd(W_e,params,data, lr, epochs):
    for i in range(epochs):
        loss, W_e = update(W_e,data,params, lr)
        print('Epoch', i+1)
        print('Loss', loss)
    return W_e
```

Throughout this part, values that we have used for the parameters are given below.

$$L_{in} = L_{out} = 16 * 16 = 256$$

$$L_{hid} = 64$$

$$\lambda = 5 * 10^{-4}$$

$$\beta = 0.01$$

$$\rho = 0.05$$

Gradient Descent with Keras

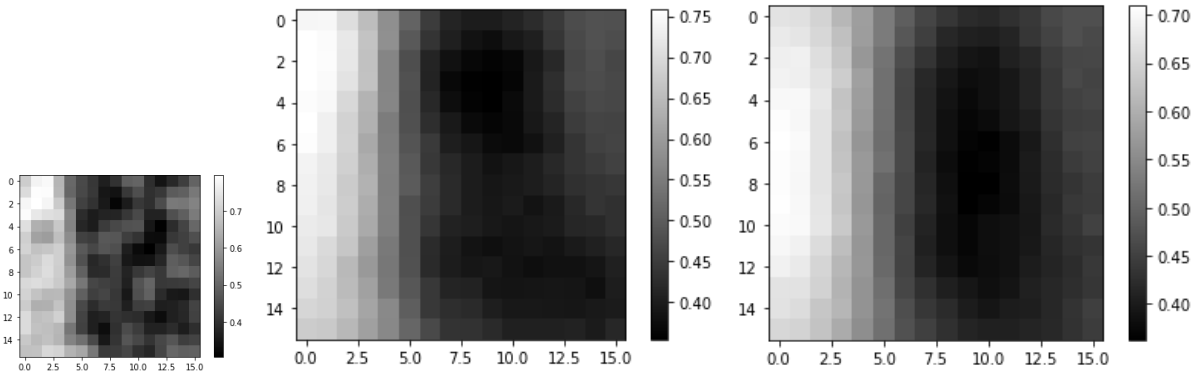
Then, we have applied the full batch gradient descent to the created network, using the same weights that we have constructed for the use of **sgd** function.

```
model.get_layer(index=0).set_weights([wMatk[0],wMatk[2].ravel()])
model.get_layer(index=1).set_weights([wMatk[1],wMatk[3].ravel()])
```

Furthermore, we have run both the "sgd" function and the Keras model.

```
model.fit(x=data_flat,y=data_flat, verbose=True, batch_size=data_flat.shape
[0], epochs=5000)
wMato = sgd(wMat,params,data_flat,0.35,5000)
```

We observe that the initial loss for both algorithms are given as 3.3388826847076416 for one instance of weight initializations. At the end of 5000 epochs the loss for self-written algorithm reached 0.9372525678465 as the loss of the Keras algorithm reached 1.0525, which are more or less the same. The difference is due to the gradient checking implementations done on the Keras backend. However, this can be considered solid proof that our aeCost implementation is correct. In order to see the difference, we have observed a test sample, the original picture, the output of the Keras model and our own model is given below.

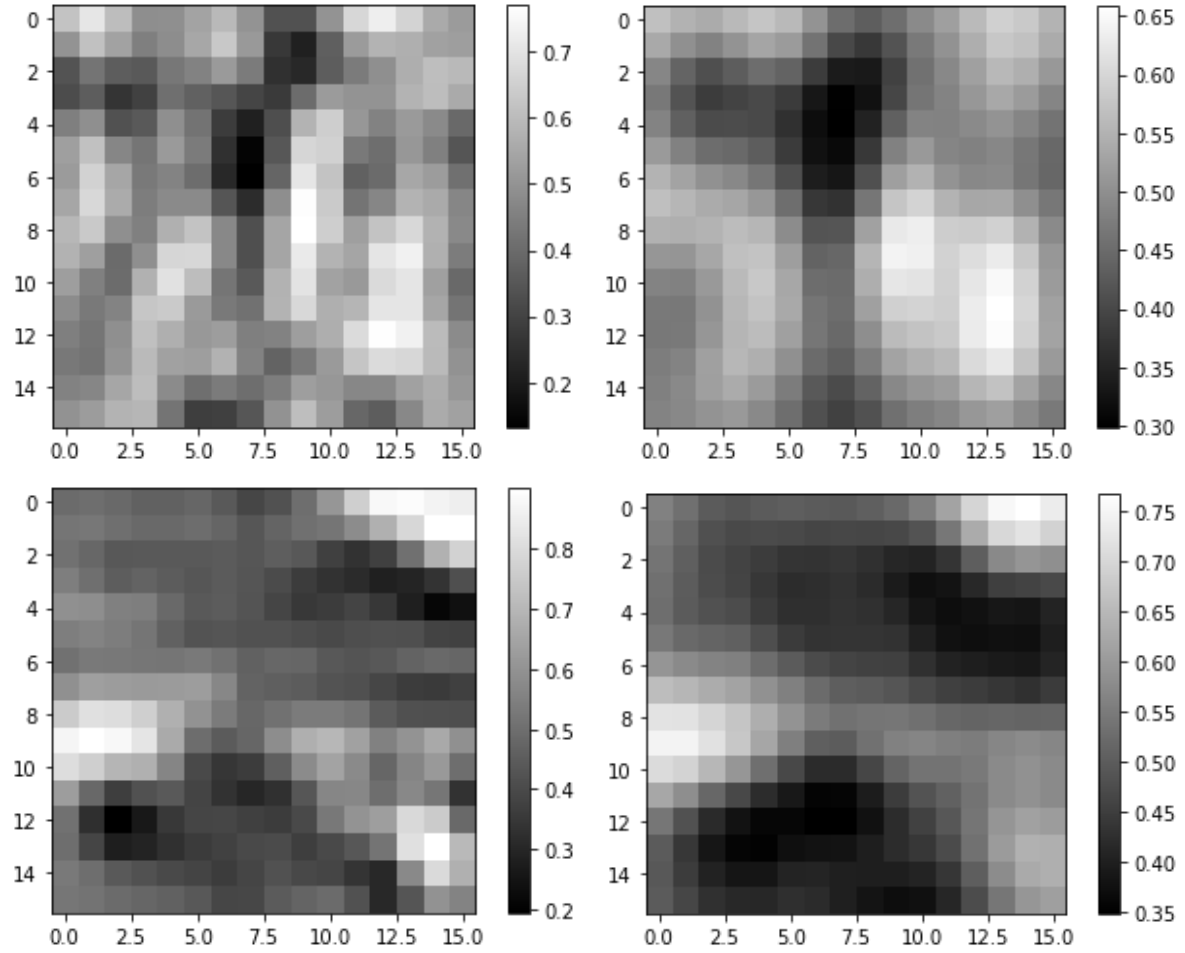


Figures 3-5 – The Sample Image and the Output of Self-Written AE and the Output of Keras AE

Here, we can see that the networks have started to learn the distributions, but still far from it. After making sure that aeCost is functional, we have continued our implementation with Keras. We have used the Adam Optimizer to minimize the loss function instead of the MATLAB optimizer function fminunc or fmincg.

Optimization Results

With the Adam optimizer the loss reached down to 0.2354 in the end of 400 epochs. Here, we have two randomly selected test samples and the outputs of the network for the samples given as input.



Figures 6-9 – Two test samples and the outputs of the optimized autoencoder from left to right.

Although different, we see that the output is similar to the input image, meaning that the network learned most of the image distribution.

Part C

This part asks us to retrieve the optimized weights and plot the hidden weights for each hidden layer neuron. The subplot is given in the next page. Here, we observe that the weight matrices resemble the natural images that are given in the dataset.

Also, we observe that the features learned the various clusters of orientations in the data. Here, we see that there are various patterns present in the data some occurring more than once. While some of them are very different from each other, we see some are closer. This may be an indication that there are enough hidden layer units present to learn the data or give more detailed information about the distribution of the data and mean that some patterns are more abundant in the data than the others. Also we see different orientations of the same patterns. For example the weight in row 7 column 1 looks is symmetric with the feature in row 6 column 7 for axis y = -x.

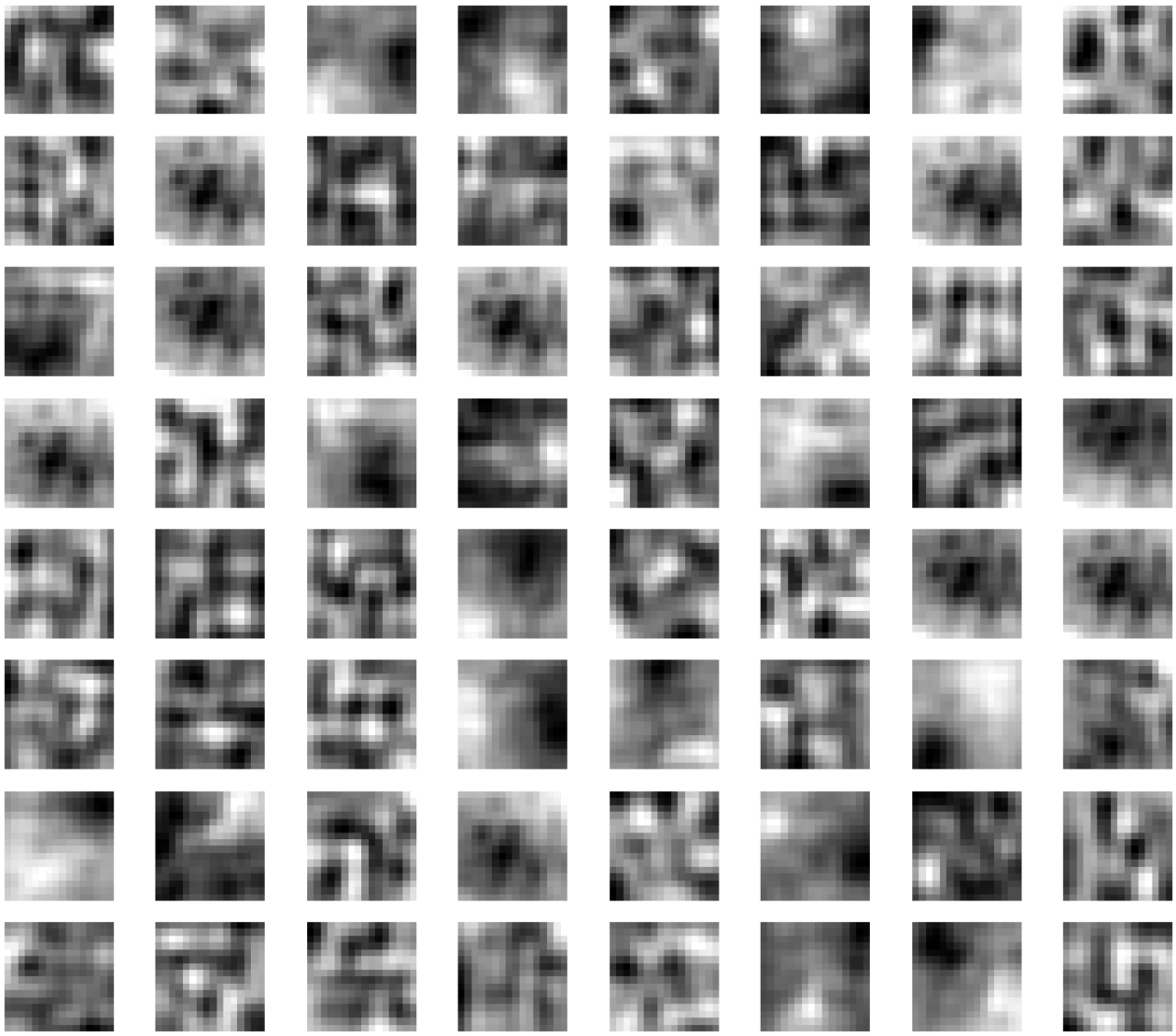


Figure 10 – Visualizations of Hidden Layer Weights.

Part D

This part asks us to implement the same network that we have in Part B and then observe the hidden weights like we have in Part C for different values of L_{hid} and λ. We are instructed to choose three different values from the intervals given below.

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

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

Hence, we have chosen the values of 10, 60 and 100 for L_{hid} and 0, 10^{-4} and 10^{-3} for λ. The visualizations and their discussion are made throughout this section.

Weights for Different Number of Hidden Layer Units

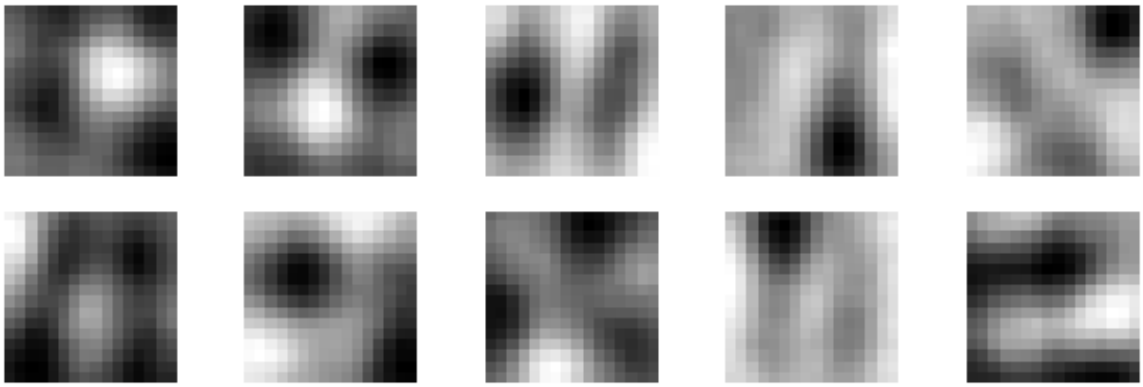


Figure 11 – Visualizations of Hidden Layer Weights for 10 Hidden Neurons.

Here, we see that the network is learning some crude approximations for the patterns in the data.

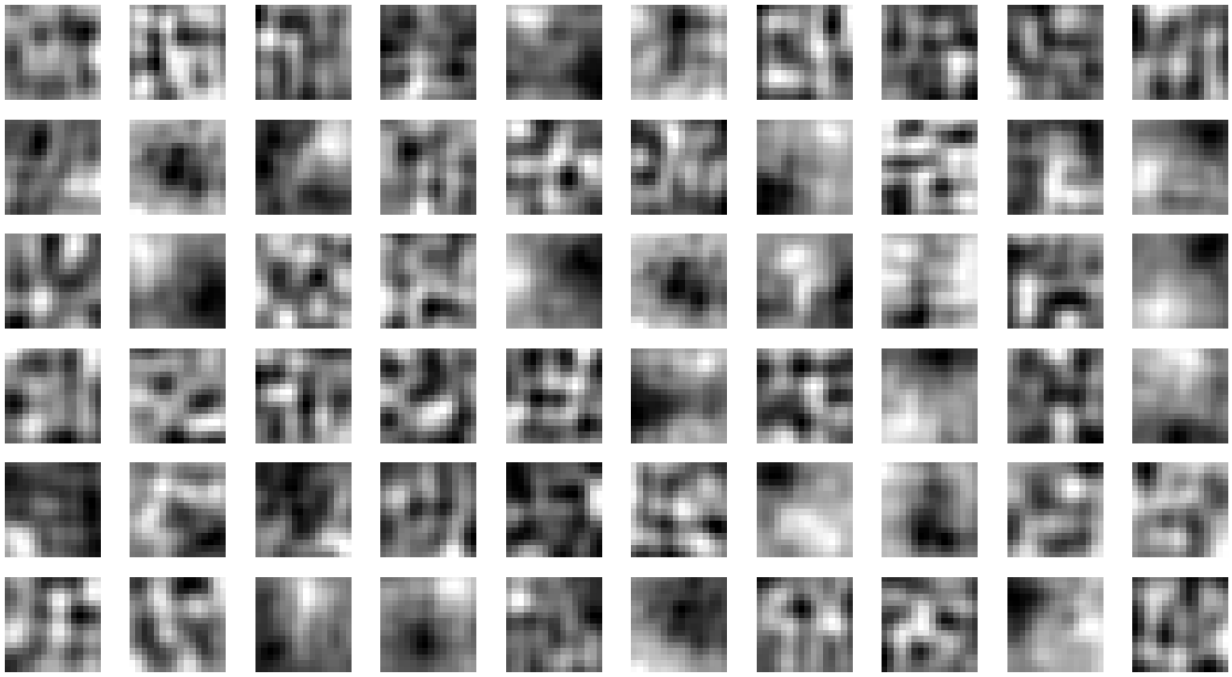


Figure 12 – Visualizations of Hidden Layer Weights for 60 Hidden Neurons.

Figure 13 – Visualizations of Hidden Layer Weights for 100 Hidden Neurons.

Throughout these three different figures, we can observe that the number of hidden layer units is an important factor in learning the overall representation of the data. We see that as the number of neurons increased, the number of learned complex patterns increased and have become more distinctive from each other. This is directly related to the learning since the network capacity increases with the increasing number of neurons. However, the capacity and accuracy are not directly related with each other since after some point, the network learns the specific details and not the overall distribution, which amounts to a type of overfitting. On the other hand, the network with 10 hidden layer units is incapable of learning the distribution at all since the capacity is not enough. The following figures are the visualizations for different λ values.

Weights for Different λ Values

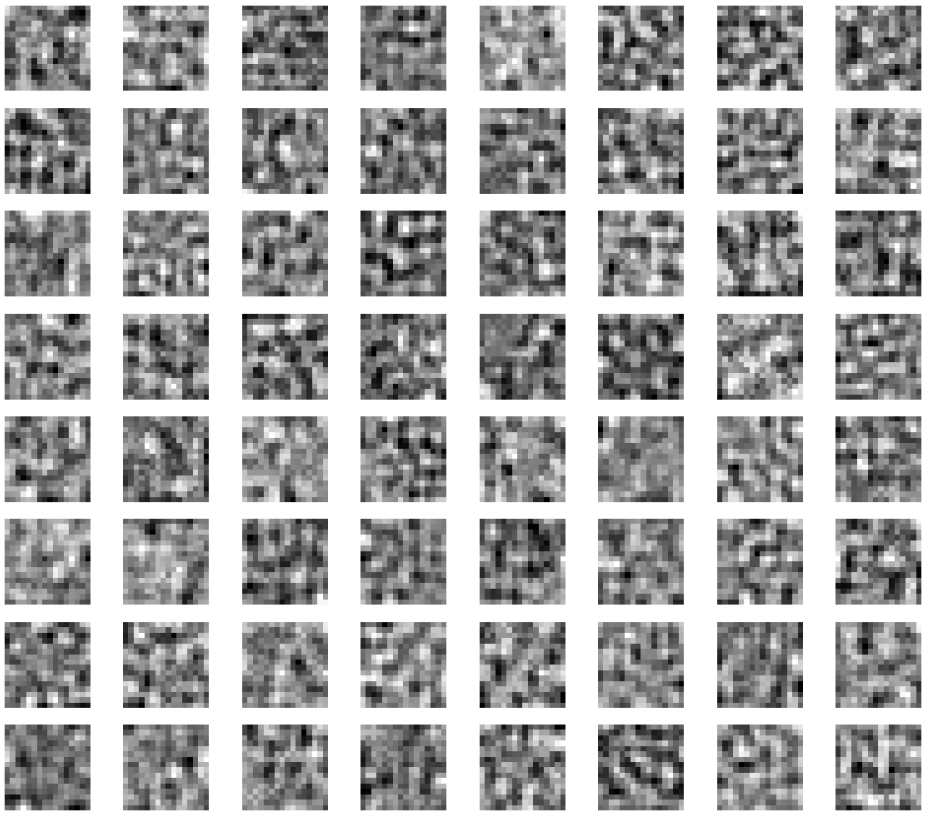


Figure 14 – Visualizations of Hidden Layer Weights for $\lambda = 0$.

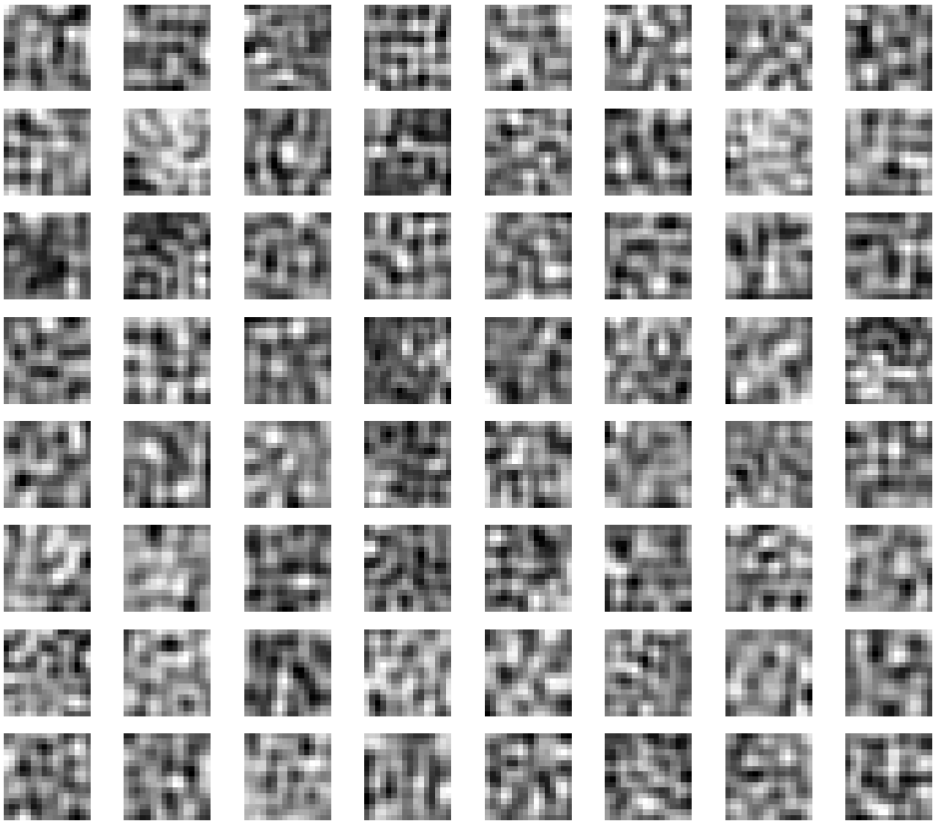


Figure 15 – Visualizations of Hidden Layer Weights for $\lambda = 10^{-4}$.

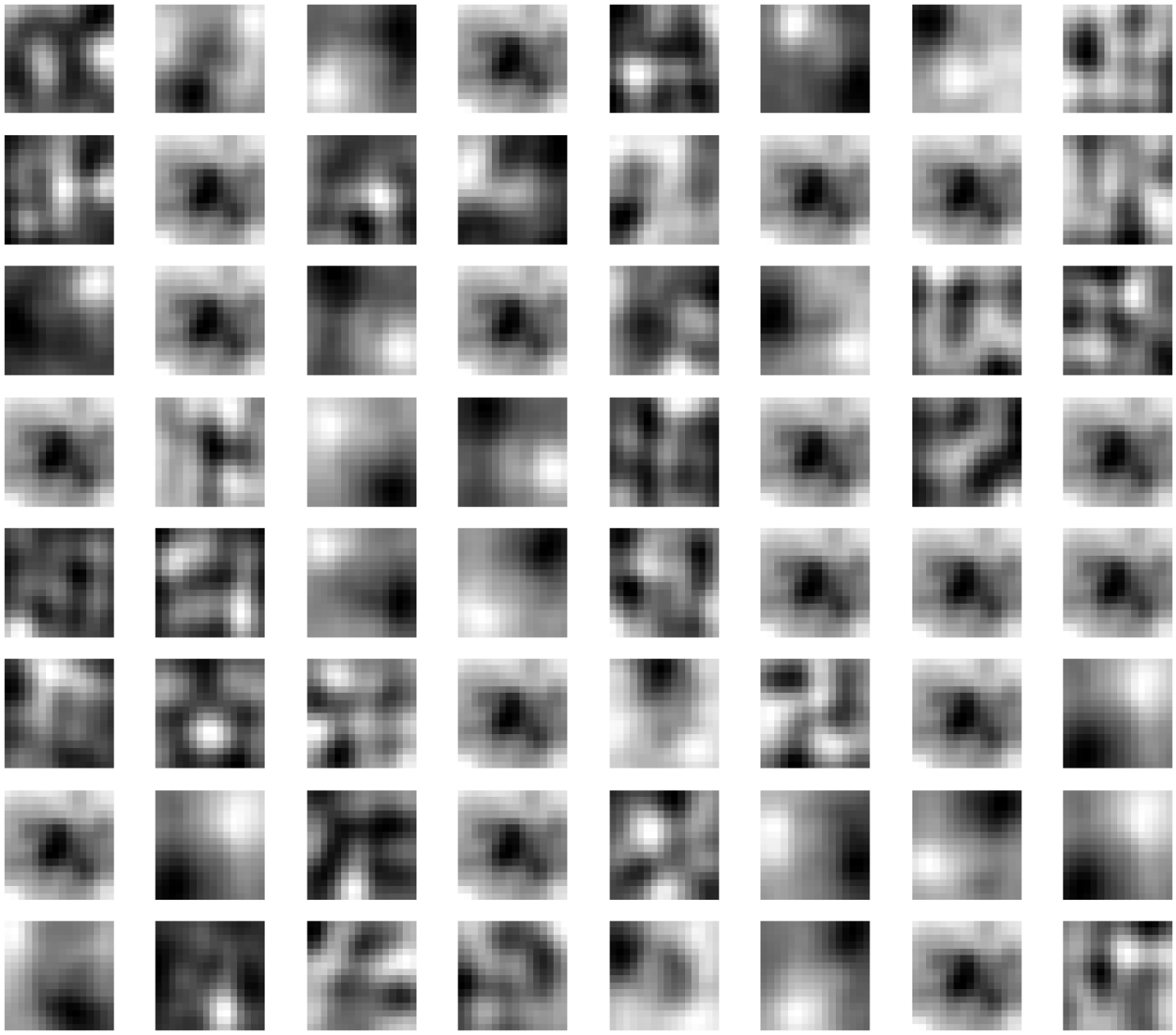


Figure 16 – Visualizations of Hidden Layer Weights for $\lambda = 10^{-3}$.

As we can observe from the Figures 14-16, as λ increases, the weights get smoother. By using weight decay in the loss, we instruct the network to look more natural and smoother. This term helps the network to learn the actual distribution of the data and not memorize the data itself by adding punishment to the layers to prevent overfitting. This way we teach the network to work better with the data it has never seen. However, if the regularization term becomes too large as seen on Figure 16, the network becomes unable to learn due to the gradient coming from the L2 regularization suppressing the gradient coming from MSE loss. Here, we believe that a medium number of hidden layer units and a medium value of λ would work the best, however, in order to clarify, we pick a test sample from the dataset and put through each of the trained network to visually compare the results. The images are given below.

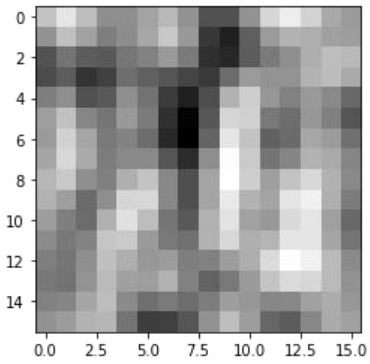
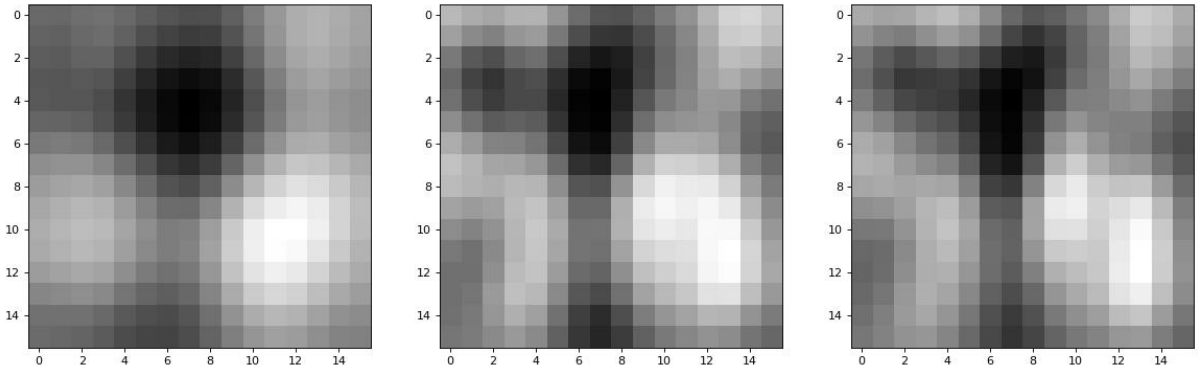
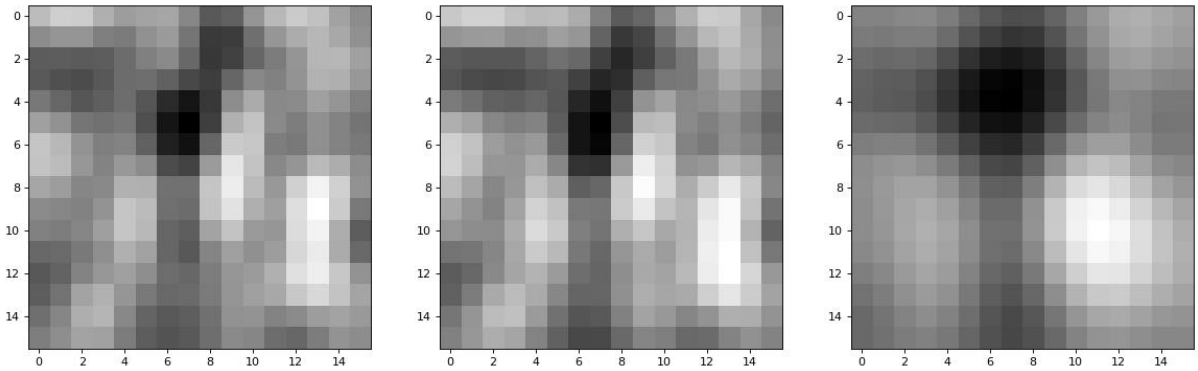


Figure 17 - Chosen Test Sample



Figures 18-20 – The Network Outputs for Hidden Layer Neurons (10,60,100)



Figures 21-23 – The Network Outputs for $\lambda = \{0, 10^{-4}, 10^{-3}\}$

In accordance with out hypothesis, the we see that as Hidden Layer Neuron size increase the capacity increases and the network adapts to the dataset better, however the network becomes easily adapted to the dataset however, at some point we see that there is little difference in between the numbers 60 and 100. Furthermore, when looking at λ values, we see that as λ increases, the network becomes more generalized, however we also see that after some point the regularization suppressing the loss and making the network unable to learn. Hence, we believe that, for generalization purposes, the medium number of hidden layer neurons and a medium value of λ is better.

Question 2

In this question, we are asked to run, observe and discuss the notebooks in the given networks named ConvolutionalNetworks.ipynb and one of either Tensorflow.ipynb or PyTorch.ipynb. The notebooks are given in Appendix B-2.

Part A

This part asked us to compile and run a demo for Convolutional Neural Networks and comment on the results. The networks run on the CIFAR-10 Image Dataset. The first part runs a naïve forward pass. There is a subsection that explains what convolution operation does to a certain grayscale image. Here, we observe that the convolution operator works as an edge detection mechanism for the given kernels. The forward pass is implemented as given below.

```
def conv_forward_naive(x, w, b, conv_param):
    """
    A naive implementation of the forward pass for a convolutional layer.

    The input consists of N data points, each with C channels, height H and
    width W. We convolve each input with F different filters, where each filter
    spans all C channels and has height HH and width WW.

    Input:
    - x: Input data of shape (N, C, H, W)
    - w: Filter weights of shape (F, C, HH, WW)
    - b: Biases, of shape (F,)
    - conv_param: A dictionary with the following keys:
      - 'stride': The number of pixels between adjacent receptive fields in the
        horizontal and vertical directions.
      - 'pad': The number of pixels that will be used to zero-pad the input.

    During padding, 'pad' zeros should be placed symmetrically (i.e equally on both sides)
    along the height and width axes of the input. Be careful not to modfiy the original
    input x directly.

    Returns a tuple of:
    - out: Output data, of shape (N, F, H', W') where H' and W' are given by
      H' = 1 + (H + 2 * pad - HH) / stride
      W' = 1 + (W + 2 * pad - WW) / stride
    - cache: (x, w, b, conv_param)
    """
    out = None

    N, C, H, W = x.shape
    F, _, HH, WW = w.shape
    stride, pad = conv_param['stride'], conv_param['pad']
    H_out = 1 + (H + 2 * pad - HH) // stride  # Use `//` for python3
    W_out = 1 + (W + 2 * pad - WW) // stride
    out = np.zeros((N, F, H_out, W_out))

    x_pad = np.pad(x, ((0,), (0,), (pad,), (pad,)), mode='constant', constant_values=0)

    for n in range(N):
       for f in range(F):
          for h_out in range(H_out):
             for w_out in range(W_out):
                out[n, f, h_out, w_out] = np.sum(
                   x_pad[n, :, h_out*stride:h_out*stride+HH,
w_out*stride:w_out*stride+WW]*w[f, :]) + b[f]
    cache = (x, w, b, conv_param)
    return out, cache
```

Here, we observe that first the dimensions of the output of the layer is evaluated according to the equation given below.

$$H' = \frac{H + 2 * Padding - H_W}{Stride}, W' = \frac{W + 2 * Padding - W_W}{Stride}$$

The H_W and W_W denotes the height and weight of the kernel. Here the batch size is conserved, and the number of output kernels is equal to the number of kernels in the convolution.

Then the demo executes a naïve backward pass that computes gradients according to the loss given by the output of a forward pass. We see that the backward pass is created with the shared gradients method. After implementation of the naïve forward and backward pass for convolutional layer, a naïve max pooling layer is formed with the forwards and the backward pass.

After naïve implementation is done, the deme continues with the fast implementations using Cython and tests the time difference between the naïve implementation. The implementation moves forward with Three-layer ConvNet. The implementation is simply based on two convolutional layers (Conv-Relu-MaxPool) and a fully connected layer. The loss is defined as the softmax loss with L2 regularizations for each layer. Then we see a method to check if the code is correct, training the model with very few training instances to check if the model overfits. After checking, the training is done with one epoch and the validation accuracy becomes 49.9%. After learning is done, the learned filters are visualized and we see that the filters learn several complex patterns and orientations in the data.

The next phase implements the batch normalization layer with its forward and backward pass. Batch normalization is a frequently used method to improve model performance by normalizing the output of the previous layer before passing it to the next. In the convolution case, each kernel is normalized in itself around all of the batches. The next part implements a Group Normalization, a technique in between the batch and layer normalizations proposed by He and Yuxin [3]. In the group normalization, the outputs of the layers are not normalized through kernels but through the batches, however in groups of G as visualized below.

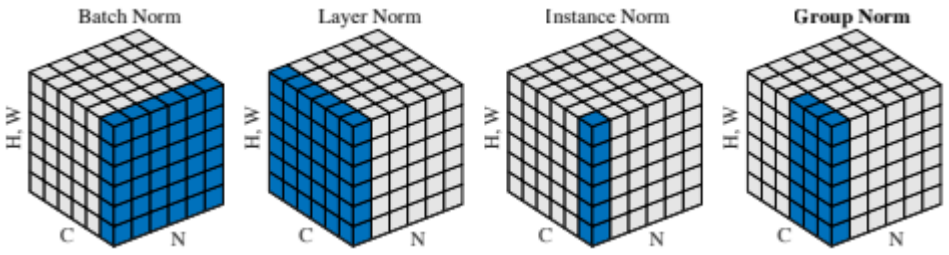


Figure 17 – Comparisons of Normalization Methods for Feature Maps [3]

Part B

This part required us to choose between the TensorFlow.ipynb and PyTorch.ipynb and I have chosen Tensorflow to work with since we most probably will run our project implementation with the Keras-Tensorflow framework. The authors explain Tensorflow as "TensorFlow is a system for executing computational graphs over Tensor objects, with native support for performing backpropagation for its Variables. In it, we work with Tensors which are n-dimensional arrays analogous to the numpy ndarray." The reason we use tensor object instead of steady numpy arrays is to be able to do GPU accelerated machine learning. One thing to note is that I have been using the recently released Tensorflow 2.0 which is highly different from the 1.0 version and hence, to not change the code to fit 2.0, I have imported Tensorflow in compatibility mode and disabled the 2.0 behavior.

The rest of the notebook consists of five parts. Loading the CIFAR-10 dataset, using Barebone Tensorflow, Keras Model API, Keras Sequential API and an Open-Ended Challenge. Unlike the PyTorch document the TODO sections of this notebook was not already coded, so I have completed

this demo up to the last part. The last part, I have referred to the completed assignment of jariasf [4].

In the first part, we have loaded the data to the notebook using Keras, since most of the popular datasets are already present in their repository. Then the Images are split to training, validation and test and all the data are normalized. After that, a simple Dataset class Is constructed that is iterable to be able to iterate over the batches.

Part two is concerned with the barebone TensorFlow implementation. In TensorFlow, the convolutional map features are hold in tensors of shape N x H x W x C, where N is the batch size, H is height, W is weights and C represents the channel. The demo starts with the implementation of a basic flatten layer and its test. In test, only placeholders that will contain the tensors when the program runs exist. Then, after the graph is constructed a session is initiated and the test is done. The next part is a two-layer network implementation with two fully connected layers and ReLU activations. After coding the forward pass, the test is done in the same way as in flatten. Then, I have completed the implementation for the three layer ConvNet forward pass as given below.

```
def three_layer_convnet(x, params):
    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.                #
    ################################################################################
    #layer1
    paddings = tf.constant([[0,0], [2,2], [2,2], [0,0]])
    x = tf.pad(x, paddings, 'CONSTANT')
    conv1 = tf.nn.conv2d(x, conv_w1, strides=[1,1,1,1], padding="VALID")+conv_b1
    relu1 = tf.nn.relu(conv1)
    #layer2
    paddings = tf.constant([[0,0], [1,1], [1,1], [0,0]])
    x2 = tf.pad(conv1, paddings, 'CONSTANT')
    conv2 = tf.nn.conv2d(x2, conv_w2, strides=[1,1,1,1], padding="VALID")+conv_b2
    relu2 = tf.nn.relu(conv2)
    #layer3
    relu2 = flatten(relu2)
    scores = tf.matmul(relu2, fc_w) + fc_b
    ################################################################################
    #                                 END OF YOUR CODE                             #
    ################################################################################
    return scores
```

Then I have used the test to check the validity of the dimensions, which have turned out to be true. Then the training step is implemented, the gradients are calculated, and gradient descent is applied. After training step, the training loop is constructed. The graph is constructed for forward and backward passes and the graph is run in a session in the loop. Then the network is run with parameters initialized with Kaiming Initialization [5]. Kaiming Initialization is as follows.

$$w_{l,i} \in \mathcal{W}_\ell, w_{l,i} \sim \mathcal{N}\left(0, \sqrt{2/n_l}\right), \forall i = 1, \dots, n_l$$

Where l is that layer and n described the number of units in that layer. After one epoch, the accuracy has reached 44.6% with the two-layer network. Then I have implemented the initialization as done in the two-layer network.

```
def three_layer_convnet_init():
    params = None
    ################################################################################
    # TODO: Initialize the parameters of the three-layer network.                  #
    ################################################################################
```

```
    conv_w1 = tf.Variable(kaiming_normal([5,5,3,32]))
    conv_b1 = tf.Variable(np.zeros([32]), dtype=tf.float32)
    conv_w2 = tf.Variable(kaiming_normal([3,3,32,16]))
    conv_b2 = tf.Variable(np.zeros([16]), dtype=tf.float32)
    fc_w = tf.Variable(kaiming_normal([32*32*16,10]))
    fc_b = tf.Variable(np.zeros([10]), dtype=tf.float32)
    params = (conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b)
    ################################################################################
    #                                END OF YOUR CODE                              #
    ################################################################################
    return params
```

After one epoch of training, the accuracy has reached 51.6%. This completes the implementation of the barebone TensorFlow.

The third part of the demo is about the use of Model API of Keras. In Model API, we define a class for the model and add the layers which are implemented by Keras in the constructor. Then we implement the call function of the class to forward pass the data. After understanding how the two-layer implementation works, I have implemented the three-layer ConvNet using the Model API as follows.

```
class ThreeLayerConvNet(tf.keras.Model):
    def __init__(self, channel_1, channel_2, num_classes):
        super().__init__()
        ########################################################################
        # TODO: Implement the __init__ method for a three-layer ConvNet. You   #
        # should instantiate layer objects to be used in the forward pass.     #
        ########################################################################
        init = tf.variance_scaling_initializer(scale=2)
        self.conv1 = tf.layers.Conv2D(channel_1, [5,5], [1,1], padding='valid', kernel_initial
izer= init, activation=tf.nn.relu)
        self.conv2 = tf.layers.Conv2D(channel_2, [3,3], [1,1], padding='valid', kernel_initial
izer= init, activation=tf.nn.relu)
        self.fc = tf.layers.Dense(num_classes, kernel_initializer=init)
        ########################################################################
        #                            END OF YOUR CODE                          #
        ########################################################################

    def call(self, x, training=None):
        scores = None
        ########################################################################
        # TODO: Implement the forward pass for a three-layer ConvNet. You      #
        # should use the layer objects defined in the __init__ method.         #
        ########################################################################
        x = tf.pad(x, tf.constant([[0,0],[2,2],[2,2],[0,0]]), 'CONSTANT')
        x = self.conv1(x)
        x = tf.pad(x, tf.constant([[0,0],[1,1],[1,1],[0,0]]), 'CONSTANT')
        x = self.conv2(x)
        x = tf.layers.flatten(x)
        scores = self.fc(x)
        ########################################################################
        #                            END OF YOUR CODE                          #
        ########################################################################
        return scores
```

Apart from barebone tf, here we observed that an optimizer object is used instead of manually implementing gradient descent in the training loop. Then we have run the code that optimizes the model according to the optimizer object, in this case was gradient descent optimizer.

```
learning_rate = 3e-3
channel_1, channel_2, num_classes = 32, 16, 10

def model_init_fn(inputs, is_training):
    model = None
    ############################################################################
    # TODO: Complete the implementation of model_fn.                           #
    ############################################################################
    model = ThreeLayerConvNet(channel_1, channel_2, num_classes)
    ############################################################################
    #                            END OF YOUR CODE                              #
    ############################################################################
```

```
    return model(inputs)

def optimizer_init_fn():
    optimizer = None
    ################################################################################
    # TODO: Complete the implementation of model_fn.                               #
    ################################################################################
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    ################################################################################
    #                                END OF YOUR CODE                              #
    ################################################################################
    return optimizer

train_part34(model_init_fn, optimizer_init_fn)
```

At the end, we were able to acquire 45% accuracy. Then the demo moves forward to Sequential API. Sequential API is the method that is easiest to use in all three, however the flexibility of the API is quite low. The sequential API only works when the layers are stacked at the end of each other. Hence, the demo shows the two-layer network that we have worked on before this time with the Sequential. Then, I have implemented the three-layer ConvNet using that. The code is given below.

```
def model_init_fn(inputs, is_training):
    model = None
    ################################################################################
    # TODO: Construct a three-layer ConvNet using tf.keras.Sequential.             #
    ################################################################################
    inpDim = (32,32,3)
    channel_1 = 32
    channel_2 = 16
    num_classes = 10
    init = tf.variance_scaling_initializer(scale=2)
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.InputLayer(input_shape = inpDim))
    model.add(tf.keras.layers.Conv2D(channel_1, [5,5], [1,1], padding = 'SAME',\
                                     kernel_initializer = init,\
                                     activation=tf.nn.relu))
    model.add(tf.keras.layers.Conv2D(channel_2, [5,5], [1,1], padding = 'SAME',\
                                     kernel_initializer = init,\
                                     activation=tf.nn.relu))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(num_classes, kernel_initializer = init))
    ################################################################################
    #                                END OF YOUR CODE                              #
    ################################################################################
    return model(inputs)

learning_rate = 5e-4
def optimizer_init_fn():
    optimizer = None
    ################################################################################
    # TODO: Complete the implementation of model_fn.                               #
    ################################################################################
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9,\
                                           use_nesterov=True)
    ################################################################################
    #                                END OF YOUR CODE                              #
    ################################################################################
    return optimizer

train_part34(model_init_fn, optimizer_init_fn)
```

In the end we observe 50.20% accuracy after training in one epoch. The next and final part of the demo is an open ended challenge. Here, I have copied the architecture run by jariasf which can be summarized as follows.

(Conv(3x3) – BatchNorm – Elu – Dropout – MaxPool(2x2))^2 – Conv(3x3) – BatchNorm – Elu – Dropout – AvgPooling – Flatten – (Dense)^3

Appendix A – References

[1] A. Ng, “Sparse Autoencoder,” *CS294A Lecture Notes*.

[2] “Deep Learning Tutorial - Sparse Autoencoder,” *Deep Learning Tutorial - Sparse Autoencoder · Chris McCormick*, 30-May-2014. [Online]. Available: https://mccormickml.com/2014/05/30/deep-learning-tutorial-sparse-autoencoder/. [Accessed: 12-Dec-2019].

[3] Y. Wu and K. He, “Group Normalization,” *International Journal of Computer Vision*, 2019.

[4] Jariasf, “jariasf/CS231n,” *GitHub*. [Online]. Available: https://github.com/jariasf/CS231n/blob/master/assignment2/TensorFlow.ipynb. [Accessed: 13-Dec-2019].

[5] K. He, X. Zhang, S. Ren and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," 2015 IEEE International Conference on Computer Vision (ICCV), Santiago, 2015, pp. 1026-1034.

Appendix B1 – Python Code for Question 1

```
#!/usr/bin/env python
# coding: utf-8

# # Question 1 - AutoEncoder NN

# #### Dependencies
# 1. Tensorflow 2.0
# 2. numpy
# 3. h5py
# 4. matplotlib

# In[3]:


import warnings
warnings.filterwarnings("ignore")
import numpy as np
import matplotlib.pyplot as plt
import h5py


# In[4]:


filename = 'assign3_data1.h5'

with h5py.File(filename, 'r') as f:
    # List all groups
    print("Keys: %s" % f.keys())
    # Get the data
    data = f[list(f.keys())[0]].value
    invXForm = f[list(f.keys())[1]].value
    xForm = f[list(f.keys())[2]].value


# ## Part A

# In[5]:


print('Data:', data.shape)
data_gs = data[:,0,:,:]*0.2126 + data[:,1,:,:]*0.7152 + data[:,2,:,:]*0.0722
print('Data_GS:', data_gs.shape)

mean = np.mean(data_gs, axis=(1,2))
print('Mean',mean.shape)
for i in range(len(mean)):
    data_gs[i,:,:] -= mean[i]

std = np.std(data_gs)
print('Std',std)

data_nor = np.minimum(data_gs, 3*std)
data_nor = np.maximum(data_nor, -3*std)

minScale = 0.10
maxScale = 0.90
data_nor = data_nor*(maxScale-minScale)/(2*np.max(data_nor))
print(np.min(data_nor))
print(np.max(data_nor))
data_nor += (minScale - np.min(data_nor))
print(np.min(data_nor))
print(np.max(data_nor))


# In[6]:
```

```
row = 20
col = 20
numPics = data_nor.shape[0]
randInd = np.random.permutation(numPics)
fig=plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
for i in range(row*col):
    plt.subplot(row,col,i+1)
    plt.imshow(np.transpose(data[randInd[i],:,:,:], (1,2,0)))
    plt.axis('off')
plt.show()

fig=plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
for i in range(row*col):
    plt.subplot(row,col,i+1)
    plt.imshow(data_nor[randInd[i],:,:], cmap='gray')
    plt.axis('off')
plt.show()


# ## Part B

# In[13]:


imWH = data_nor.shape[1]
data_flat = np.reshape(data_nor, (data_nor.shape[0],imWH**2))
print(data_flat.shape)

def w_0(Lpre, Lpost):
    return np.sqrt(6/(Lpre+Lpost))

#Lin=Lout written separately just for formality.
def defWeights(Lin, Lhid, Lout):
    #random seed is conserved in order to observe the same weight matrices in both
networks
    np.random.seed(45)
    W1 = np.random.uniform(-w_0(Lin,Lhid),w_0(Lin,Lhid),(Lin,Lhid))
    b1 = np.random.uniform(-w_0(Lin,Lhid),w_0(Lin,Lhid),(1,Lhid))
    W2 = np.random.uniform(-w_0(Lhid,Lout),w_0(Lhid,Lout),(Lhid,Lout))
    b2 = np.random.uniform(-w_0(Lhid,Lout),w_0(Lhid,Lout),(1,Lout))
    return (W1, W2, b1, b2)

Lin = Lout = imWH**2
Lhid = 64
lmb = 5e-4
beta = 0.01
rho = 0.05
params = (Lin, Lhid, lmb, beta, rho)


# In[39]:


def sigmoid(x):
    expx = np.exp(x)
    return expx/(1+expx)

def derSigmoid(x):
    return x*(1-x);

def forwardPass(W_e, data):
    W1, W2, b1, b2 = W_e
    W1_e = np.append(W1,b1, axis=0)
    W2_e = np.append(W2,b2, axis=0)

    #input -> hidden
```

```
    data_ = np.append(data, np.ones((data.shape[0], 1)), axis=1)
    z = sigmoid(np.matmul(data_,W1_e))

    #hidden -> output
    z_ = np.append(z, np.ones((z.shape[0], 1)), axis=1)
    out = sigmoid(np.matmul(z_,W2_e))

    return z, out

def aeCost(W_e, data, params):
    (Lin, Lhid, lmb, beta, rho) = params
    W1, W2, b1, b2 = W_e
    N = data.shape[0]

    z, dataRec = forwardPass(W_e, data)
    z_mean = np.mean(z, axis=0)
    #print('z', z.shape)
    #print('zmean', z_mean.shape)

    #print(dataRec)
    J1 = (1/(2*N))*np.sum(np.power((data - dataRec),2))
    J2 = (lmb/2)*(np.sum(W1**2) + np.sum(W2**2))
    kl1 = rho*np.log(z_mean/rho)
    kl2 = (1-rho)*np.log((1-z_mean)/(1-rho))
    J3 =  beta*np.sum(kl1+kl2)
    J = J1 + J2 - J3

    deltaOut = -(data-dataRec)*derSigmoid(dataRec)

    derKL = np.tile(beta*(-(rho/z_mean.T)+((1-rho)/(1-z_mean.T))), (10240,1)).T

    deltaHid = (np.matmul(W2,deltaOut.T)+ derKL) * derSigmoid(z).T

    gradWout = (1/N)*(np.matmul(deltaOut.T,z).T + lmb*W2)
    gradBout = np.mean(deltaOut, axis=0)

    gradWhid = (1/N)*(np.matmul(data.T,deltaHid.T) + lmb*W1)
    gradBhid = np.mean(deltaHid, axis=1)
    return J, (gradWhid, gradWout, gradBhid, gradBout)


# In[218]:


def update(W_e, data, params, learning_rate):
    (Lin, Lhid, lmb, beta, rho) = params
    W1, W2, b1, b2 = W_e
    N = data.shape[0]

    z, dataRec = forwardPass(W_e, data)
    z_mean = np.mean(z, axis=0)

    J1 = (1/(2*N))*np.sum(np.power((data - dataRec),2))
    J2 = (lmb/2)*(np.sum(W1**2) + np.sum(W2**2))
    kl1 = rho*np.log(z_mean/rho)
    kl2 = (1-rho)*np.log((1-z_mean)/(1-rho))
    J3 =  beta*np.sum(kl1+kl2)
    J = J1 + J2 - J3



    deltaOut = -(data-dataRec)*derSigmoid(dataRec)

    derKL = np.tile(beta*(-(rho/z_mean.T)+((1-rho)/(1-z_mean.T))), (10240,1)).T

    deltaHid = (np.matmul(W2,deltaOut.T)+ derKL) * derSigmoid(z).T

    gradWout = (1/N)*(np.matmul(deltaOut.T,z).T + lmb*W2)
```

```
    gradBout = np.mean(deltaOut, axis=0)

    gradWhid = (1/N)*(np.matmul(data.T,deltaHid.T) + lmb*W1)
    gradBhid = np.mean(deltaHid, axis=1)

    W1 -= gradWhid*learning_rate
    W2 -= gradWout*learning_rate
    b1 -= gradBhid*learning_rate
    b2 -= gradBout*learning_rate
    return J, (W1,W2,b1,b2)


def sgd(W_e,params,data, lr, epochs):
    for i in range(epochs):
        loss, W_e = update(W_e,data,params, lr)
        print('Epoch', i+1)
        print('Loss', loss)
    return W_e

wMat = defWeights(Lin, Lhid, Lin)
(loss, gr) = aeCost(wMat,data_flat,params)
print(loss)
wMato = sgd(wMat,params,data_flat,0.35,5000)


# We will now use keras to implement the same network since python does not have an
implicit gradient descent solver implemented. Hence, we will create the network,
run one forward pass and show that the gradients are more or less similar with the
gradients that we have manually computed.

# In[42]:


import tensorflow as tf
from tensorflow.keras import initializers, losses, regularizers, optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import backend as K


# In[43]:


def kl(rho, beta, N):
    def customKL(out):
        kl1 = rho*K.log(rho/K.mean(out, axis=0))
        kl2 = (1-rho)*K.log((1-rho)/(1-K.mean(out, axis=0)))
        return N*beta*K.sum(kl1+kl2)
    return customKL

def mse():
    def customMSE(y_true, y_pred):
        return 0.5*K.sum(K.mean(K.square(y_true-y_pred),axis=0))
    return customMSE


# In[219]:


model = Sequential()
model.add(Dense(Lhid, input_shape=(data_flat.shape[1],), activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2),  activity_regularizer=kl(rho, beta,
data_flat.shape[0])))

model.add(Dense(Lout, activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2)))

opt = optimizers.SGD(learning_rate=0.35, momentum=0.0, nesterov=False)
```

```
model.compile(optimizer=opt, loss=mse())

wMatk = defWeights(Lin,Lhid,Lout)
model.get_layer(index=0).set_weights([wMatk[0],wMatk[2].ravel()])
model.get_layer(index=1).set_weights([wMatk[1],wMatk[3].ravel()])

loss = model.evaluate(x=data_flat,y=data_flat,verbose=False,
batch_size=data_flat.shape[0])
print(loss)


# In[220]:


model.fit(x=data_flat,y=data_flat, verbose=True, batch_size=data_flat.shape[0],
epochs=5000)


# In[221]:


testInd = 15
(hid,pred) = forwardPass(wMato, data_flat)
print('Original Image')
plt.imshow(data_nor[testInd,:].reshape(imWH,imWH), cmap='gray'),plt.colorbar()
plt.show()
print('SGD w/aeCost')
plt.imshow(pred[testInd,:].reshape((imWH,imWH)), cmap='gray'),plt.colorbar()
plt.show()
print('SGD w/Keras')
encoded = model.predict(data_flat[testInd,:].reshape((imWH**2,1)).T)
plt.imshow(encoded.reshape((imWH,imWH)), cmap='gray'),plt.colorbar()
plt.show()


# In[173]:


Lin = Lout = imWH**2
Lhid = 64
lmb = 5e-4
beta = 0.01
rho = 0.05

model_ = Sequential()
model_.add(Dense(Lhid, input_shape=(data_flat.shape[1],), activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2),  activity_regularizer=kl(rho, beta,
data_flat.shape[0])))

model_.add(Dense(Lout, activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2)))

model_.compile(optimizer='Adam', loss=mse())

wMatk_ = defWeights(Lin, Lhid, Lout)
model_.get_layer(index=0).set_weights([wMatk_[0],wMatk_[2].ravel()])
model_.get_layer(index=1).set_weights([wMatk_[1],wMatk_[3].ravel()])

loss = model_.evaluate(x=data_flat,y=data_flat,verbose=False,
batch_size=data_flat.shape[0])
print(loss)


# In[174]:


model_.fit(x=data_flat, y=data_flat, verbose=True, batch_size=data.shape[0],
epochs=5000)
```

```
# In[223]:


testInd = 500

plt.imshow(data_nor[testInd,:].reshape(imWH,imWH), cmap='gray'),plt.colorbar()
plt.show()
encoded = model_.predict(data_flat[testInd,:].reshape((imWH**2,1)).T)
plt.imshow(encoded.reshape((imWH,imWH)), cmap='gray'),plt.colorbar()
plt.show()


# ## Part C

# In[176]:


hidWeights = model_.get_layer(index=0).trainable_weights[0]
print(hidWeights.shape)
print(hidWeights.numpy)


# In[224]:


row = 8
col = 8
fig=plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
for i in range(hidWeights.shape[1]):
    plt.subplot(row,col,i+1)
    plt.imshow(np.reshape(hidWeights[:,i],(imWH,imWH)), cmap='gray',
interpolation='none')
    plt.axis('off')
plt.show()


# ## Part D
# Hidden

# In[137]:


Lin = Lout = imWH**2
Lhid = 10
lmb = 5e-4
beta = 0.01
rho = 0.05
params = (Lin, Lhid, lmb, beta, rho)

model1 = Sequential()
model1.add(Dense(Lhid, input_shape=(data_flat.shape[1],), activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2),            activity_regularizer=kl(rho,
beta, data_flat.shape[0])))

model1.add(Dense(Lout, activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2)))
model1.compile(optimizer='Adam', loss=mse())

wMatk_ = defWeights(Lin, Lhid, Lout)
model1.get_layer(index=0).set_weights([wMatk_[0],wMatk_[2].ravel()])
model1.get_layer(index=1).set_weights([wMatk_[1],wMatk_[3].ravel()])

model1.fit(x=data_flat,y=data_flat, verbose=True, batch_size=data_flat.shape[0],
epochs=5000)
```

```
# In[180]:


hidWeights1 = model1.get_layer(index=0).trainable_weights[0]
row = 2
col = 5
fig=plt.figure(figsize=(18, 6), dpi= 40, facecolor='w', edgecolor='k')
for i in range(hidWeights1.shape[1]):
    plt.subplot(row,col,i+1)
    plt.imshow(np.reshape(hidWeights1[:,i],(imWH,imWH)), cmap='gray')
    plt.axis('off')
plt.show()


# In[141]:


Lin = Lout = imWH**2
Lhid = 60
lmb = 5e-4
beta = 0.01
rho = 0.05
params = (Lin, Lhid, lmb, beta, rho)

model2 = Sequential()
model2.add(Dense(Lhid, input_shape=(data_flat.shape[1],), activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2),            activity_regularizer=kl(rho,
beta, data_flat.shape[0])))

model2.add(Dense(Lout, activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2)))
model2.compile(optimizer='Adam', loss=mse())

wMatk_ = defWeights(Lin, Lhid, Lout)
model2.get_layer(index=0).set_weights([wMatk_[0],wMatk_[2].ravel()])
model2.get_layer(index=1).set_weights([wMatk_[1],wMatk_[3].ravel()])

model2.fit(x=data_flat,y=data_flat, verbose=True, batch_size=data_flat.shape[0],
epochs=5000)


# In[142]:


hidWeights2 = model2.get_layer(index=0).trainable_weights[0]
row = 6
col = 10
fig=plt.figure(figsize=(18, 10), dpi= 80, facecolor='w', edgecolor='k')
for i in range(hidWeights2.shape[1]):
    plt.subplot(row,col,i+1)
    plt.imshow(np.reshape(hidWeights2[:,i],(imWH,imWH)), cmap='gray')
    plt.axis('off')
plt.show()


# In[181]:


Lin = Lout = imWH**2
Lhid = 100
lmb = 5e-4
beta = 0.01
rho = 0.05
params = (Lin, Lhid, lmb, beta, rho)

model3 = Sequential()
model3.add(Dense(Lhid, input_shape=(data_flat.shape[1],), activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2),  activity_regularizer=kl(rho, beta,
```

```
data_flat.shape[0])))

model3.add(Dense(Lout, activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2)))
model3.compile(optimizer='Adam', loss=mse())

wMatk_ = defWeights(Lin, Lhid, Lout)
model3.get_layer(index=0).set_weights([wMatk_[0],wMatk_[2].ravel()])
model3.get_layer(index=1).set_weights([wMatk_[1],wMatk_[3].ravel()])


model3.fit(x=data_flat,y=data_flat, verbose=True, batch_size=data_flat.shape[0],
epochs=5000)


# In[182]:


hidWeights3 = model3.get_layer(index=0).trainable_weights[0]
row = 10
col = 10
fig=plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
for i in range(hidWeights3.shape[1]):
    plt.subplot(row,col,i+1)
    plt.imshow(np.reshape(hidWeights3[:,i],(imWH,imWH)), cmap='gray')
    plt.axis('off')
plt.show()


# lambda

# In[183]:


Lin = Lout = imWH**2
Lhid = 64
lmb = 0
beta = 0.01
rho = 0.05
params = (Lin, Lhid, lmb, beta, rho)

model4 = Sequential()
model4.add(Dense(Lhid, input_shape=(data_flat.shape[1],), activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2),
activity_regularizer=kl(rho, beta, data_flat.shape[0])))

model4.add(Dense(Lout, activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2)))
model4.compile(optimizer='Adam', loss=mse())

wMatk_ = defWeights(Lin, Lhid, Lout)
model4.get_layer(index=0).set_weights([wMatk_[0],wMatk_[2].ravel()])
model4.get_layer(index=1).set_weights([wMatk_[1],wMatk_[3].ravel()])

model4.fit(x=data_flat,y=data_flat, verbose=True, batch_size=data_flat.shape[0],
epochs=5000)


# In[189]:


hidWeights4 = model4.get_layer(index=0).trainable_weights[0]
row = 8
col = 8
fig=plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
for i in range(hidWeights4.shape[1]):
    plt.subplot(row,col,i+1)
    plt.imshow(np.reshape(hidWeights4[:,i],(imWH,imWH)), cmap='gray')
```

```
    plt.axis('off')
plt.show()


# In[185]:


Lin = Lout = imWH**2
Lhid = 64
lmb = 1e-4
beta = 0.01
rho = 0.05
params = (Lin, Lhid, lmb, beta, rho)

model5 = Sequential()
model5.add(Dense(Lhid, input_shape=(data_flat.shape[1],), activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2),  activity_regularizer=kl(rho, beta,
data_flat.shape[0])))

model5.add(Dense(Lout, activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2)))
model5.compile(optimizer='Adam', loss=mse())

wMatk_ = defWeights(Lin, Lhid, Lout)
model5.get_layer(index=0).set_weights([wMatk_[0],wMatk_[2].ravel()])
model5.get_layer(index=1).set_weights([wMatk_[1],wMatk_[3].ravel()])


model5.fit(x=data_flat,y=data_flat, verbose=True, batch_size=data_flat.shape[0],
epochs=5000)


# In[186]:


hidWeights5 = model5.get_layer(index=0).trainable_weights[0]
row = 8
col = 8
fig=plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
for i in range(hidWeights5.shape[1]):
    plt.subplot(row,col,i+1)
    plt.imshow(np.reshape(hidWeights5[:,i],(imWH,imWH)), cmap='gray')
    plt.axis('off')
plt.show()


# In[194]:


Lin = Lout = imWH**2
Lhid = 64
lmb = 1e-3
beta = 0.01
rho = 0.05
params = (Lin, Lhid, lmb, beta, rho)

model6 = Sequential()
model6.add(Dense(Lhid, input_shape=(data_flat.shape[1],), activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2),  activity_regularizer=kl(rho, beta,
data_flat.shape[0])))

model6.add(Dense(Lout, activation='sigmoid',
kernel_regularizer=regularizers.l2(lmb/2)))
model6.compile(optimizer='Adam', loss=mse())

wMatk_ = defWeights(Lin, Lhid, Lout)
model6.get_layer(index=0).set_weights([wMatk_[0],wMatk_[2].ravel()])
model6.get_layer(index=1).set_weights([wMatk_[1],wMatk_[3].ravel()])
```

```
model6.fit(x=data_flat,y=data_flat, verbose=True, batch_size=data_flat.shape[0],
epochs=5000)


# In[195]:


hidWeights6 = model6.get_layer(index=0).trainable_weights[0]
row = 8
col = 8
fig=plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
for i in range(hidWeights6.shape[1]):
    plt.subplot(row,col,i+1)
    plt.imshow(np.reshape(hidWeights6[:,i],(imWH,imWH)), cmap='gray')
    plt.axis('off')
plt.show()
```

Appendix B2 – Python Codes for 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_numerica
l_gradient
from cs231n.layers import *
from cs231n.fast_layers import *
from cs231n.solver import Solver

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

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipyt
hon
%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
```

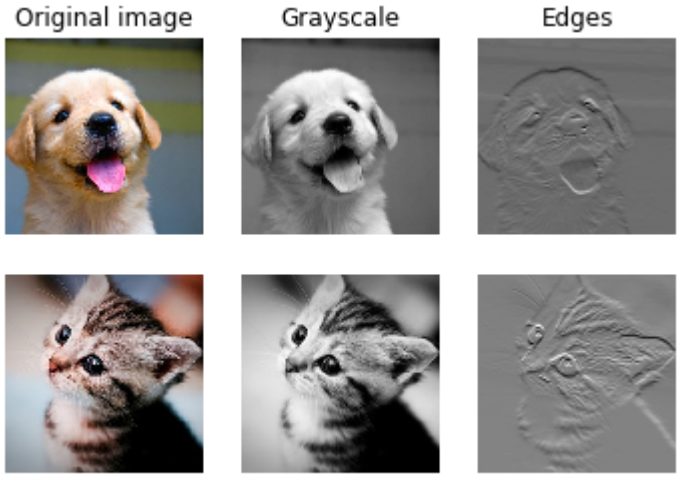
Aside: Image processing via convolutions

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 scipy.misc import imread, imresize

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

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

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

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

```
C:\Users\ayhok\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: Deprecati
onWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports u
ntil
C:\Users\ayhok\Anaconda3\lib\site-packages\ipykernel_launcher.py:10: Deprecat
ionWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.3.0.
Use Pillow instead: ``numpy.array(Image.fromarray(arr).resize())``.
  # Remove the CWD from sys.path while we load stuff.
C:\Users\ayhok\Anaconda3\lib\site-packages\ipykernel_launcher.py:11: Deprecat
ionWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.3.0.
Use Pillow instead: ``numpy.array(Image.fromarray(arr).resize())``.
  # This is added back by InteractiveShellApp.init_path()
```



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, c
onv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, c
onv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, c
onv_param)[0], b, dout)

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

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

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

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, poo
l_param)[0], x, dout)

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

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

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

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: 12.259986s
Fast: 0.043951s
Speedup: 278.949501x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 18.215454s
Fast: 0.029653s
Speedup: 614.295428x
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: 0.348665s
fast: 0.007998s
speedup: 43.596709x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.963423s
fast: 0.034217s
speedup: 28.156352x
dx difference:  0.0
```

Convolutional "sandwich" layers

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

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

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

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

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

```
Testing conv_relu_pool
dx error:  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, co
nv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, co
nv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, co
nv_param)[0], b, dout)

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

```
Testing conv_relu:
dx error:  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], verb
ose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,
grads[param_name])))
```

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

Overfit small data

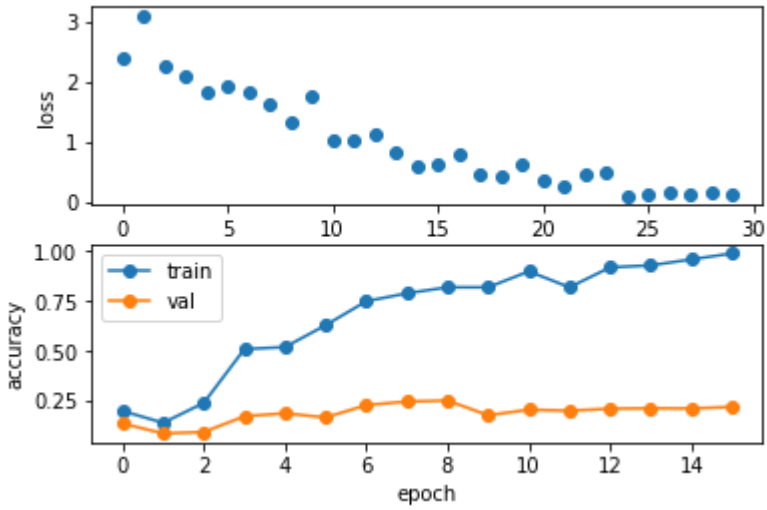
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

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:

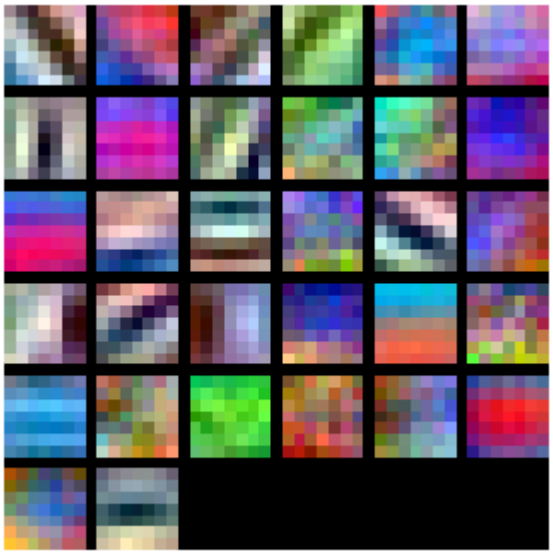
In [16]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

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. (https://arxiv.org/abs/1502.03167)

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))
```

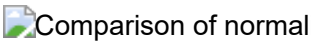
```
dx error:  3.083846820796372e-07
dgamma error:  7.09738489671469e-12
dbeta error:  3.275608725278405e-12
```

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. (https://arxiv.org/pdf/1607.06450.pdf)

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

[6] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005. (https://ieeexplore.ieee.org/abstract/document/1467360/)

Group normalization: forward

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

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

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

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

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

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

Spatial group normalization: backward

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

In [22]:
```
np.random.seed(231)
N, C, H, W = 2, 6, 4, 5
G = 2
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

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

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

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

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

In [ ]:
```
```

What's this TensorFlow 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, TensorFlow (or PyTorch, if you switch over to that notebook)

What is it?

TensorFlow is a system for executing computational graphs over Tensor objects, with native support for performing backpropogation for its Variables. In it, we work with Tensors which are n-dimensional arrays analogous to the numpy ndarray.

Why?

- Our code will now run on GPUs! Much faster training. Writing your own modules to run on GPUs is beyond the scope of this class, unfortunately.
- 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.

How will I learn TensorFlow?

TensorFlow has many excellent tutorials available, including those from Google themselves (https://www.tensorflow.org/get_started/get_started).

Otherwise, this notebook will walk you through much of what you need to do to train models in TensorFlow. See the end of the notebook for some links to helpful tutorials if you want to learn more or need further clarification on topics that aren't fully explained here.

Table of Contents

This notebook has 5 parts. We will walk through TensorFlow at three different levels of abstraction, which should help you better understand it and prepare you for working on your project.

1. Preparation: load the CIFAR-10 dataset.
2. Barebone TensorFlow: we will work directly with low-level TensorFlow graphs.
3. Keras Model API: we will use `tf.keras.Model` to define arbitrary neural network architecture.
4. Keras Sequential API: we will use `tf.keras.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
`tf.keras.Model`	High	Medium
`tf.keras.Sequential`	Low	High

Part I: Preparation

First, we load the CIFAR-10 dataset. This might take a few minutes to download the first time you run it, but after that the files should be cached on disk and loading should be faster.

In previous parts of the assignment we used CS231N-specific code to download and read the CIFAR-10 dataset; however the `tf.keras.datasets` package in TensorFlow provides prebuilt utility functions for loading many common datasets.

For the purposes of this assignment we will still write our own code to preprocess the data and iterate through it in minibatches. The `tf.data` package in TensorFlow provides tools for automating this process, but working with this package adds extra complication and is beyond the scope of this notebook. However using `tf.data` can be much more efficient than the simple approach used in this notebook, so you should consider using it for your project.

In [11]:
```
import os
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import numpy as np
import math
import timeit
import matplotlib.pyplot as plt

%matplotlib inline
```

```
WARNING: Logging before flag parsing goes to stderr.
W1213 20:21:12.512827 10688 deprecation.py:323] From c:\users\ayhok\appdata\l
ocal\programs\python\python37\lib\site-packages\tensorflow_core\python\compat
\v2_compat.py:65: disable_resource_variables (from tensorflow.python.ops.vari
able_scope) is deprecated and will be removed in a future version.
Instructions for updating:
non-resource variables are not supported in the long term
```

In [12]:
```
def load_cifar10(num_training=49000, num_validation=1000, num_test=10000):
    """
    Fetch the CIFAR-10 dataset from the web and perform preprocessing to prepa
re
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 dataset and use appropriate data types and shapes
    cifar10 = tf.keras.datasets.cifar10.load_data()
    (X_train, y_train), (X_test, y_test) = cifar10
    X_train = np.asarray(X_train, dtype=np.float32)
    y_train = np.asarray(y_train, dtype=np.int32).flatten()
    X_test = np.asarray(X_test, dtype=np.float32)
    y_test = np.asarray(y_test, dtype=np.int32).flatten()

    # Subsample the data
    mask = range(num_training, num_training + num_validation)
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = range(num_training)
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean pixel and divide by std
    mean_pixel = X_train.mean(axis=(0, 1, 2), keepdims=True)
    std_pixel = X_train.std(axis=(0, 1, 2), keepdims=True)
    X_train = (X_train - mean_pixel) / std_pixel
    X_val = (X_val - mean_pixel) / std_pixel
    X_test = (X_test - mean_pixel) / std_pixel

    return X_train, y_train, X_val, y_val, X_test, y_test


# Invoke the above function to get our data.
NHW = (0, 1, 2)
X_train, y_train, X_val, y_val, X_test, y_test = load_cifar10()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape, y_train.dtype)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,) int32
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

Preparation: Dataset object

For our own convenience we'll define a lightweight `Dataset` class which lets us iterate over data and labels. This is not the most flexible or most efficient way to iterate through data, but it will serve our purposes.

In [13]:
```
class Dataset(object):
    def __init__(self, X, y, batch_size, shuffle=False):
        """
        Construct a Dataset object to iterate over data X and labels y

        Inputs:
        - X: Numpy array of data, of any shape
        - y: Numpy array of labels, of any shape but with y.shape[0] == X.shape[0]
        - batch_size: Integer giving number of elements per minibatch
        - shuffle: (optional) Boolean, whether to shuffle the data on each epoch
        """
        assert X.shape[0] == y.shape[0], 'Got different numbers of data and labels'
        self.X, self.y = X, y
        self.batch_size, self.shuffle = batch_size, shuffle

    def __iter__(self):
        N, B = self.X.shape[0], self.batch_size
        idxs = np.arange(N)
        if self.shuffle:
            np.random.shuffle(idxs)
        return iter((self.X[i:i+B], self.y[i:i+B]) for i in range(0, N, B))


train_dset = Dataset(X_train, y_train, batch_size=64, shuffle=True)
val_dset = Dataset(X_val, y_val, batch_size=64, shuffle=False)
test_dset = Dataset(X_test, y_test, batch_size=64)
```

In [14]:
```
# We can iterate through a dataset like this:
for t, (x, y) in enumerate(train_dset):
    print(t, x.shape, y.shape)
    if t > 5: break
```

```
0 (64, 32, 32, 3) (64,)
1 (64, 32, 32, 3) (64,)
2 (64, 32, 32, 3) (64,)
3 (64, 32, 32, 3) (64,)
4 (64, 32, 32, 3) (64,)
5 (64, 32, 32, 3) (64,)
6 (64, 32, 32, 3) (64,)
```

You can optionally **use GPU by setting the flag to True below**. It's not neccessary to use a GPU for this assignment; if you are working on Google Cloud then we recommend that you do not use a GPU, as it will be significantly more expensive.

In [15]:
```
# Set up some global variables
USE_GPU = False

if USE_GPU:
    device = '/device:GPU:0'
else:
    device = '/cpu:0'

# Constant to control how often we print when training models
print_every = 100

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

```
Using device:  /cpu:0
```

Part II: Barebone TensorFlow

TensorFlow ships with various high-level APIs which make it very convenient to define and train neural networks; we will cover some of these constructs in Part III and Part IV of this notebook. In this section we will start by building a model with basic TensorFlow constructs to help you better understand what's going on under the hood of the higher-level APIs.

TensorFlow is primarily a framework for working with **static computational graphs**. Nodes in the computational graph are Tensors which will hold n-dimensional arrays when the graph is run; edges in the graph represent functions that will operate on Tensors when the graph is run to actually perform useful computation.

This means that a typical TensorFlow program is written in two distinct phases:

1. Build a computational graph that describes the computation that you want to perform. This stage doesn't actually perform any computation; it just builds up a symbolic representation of your computation. This stage will typically define one or more `placeholder` objects that represent inputs to the computational graph.
2. Run the computational graph many times. Each time the graph is run you will specify which parts of the graph you want to compute, and pass a `feed_dict` dictionary that will give concrete values to any `placeholder`s in the graph.

TensorFlow warmup: Flatten Function

We can see this in action by defining a simple `flatten` function that will reshape image data for use in a fully-connected network.

In TensorFlow, data for convolutional feature maps is typically stored in a Tensor of shape N x H x W x C where:

- N is the number of datapoints (minibatch size)
- H is the height of the feature map
- W is the width of the feature map
- C is the number of channels in the feature map

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 `H x W x C` values per representation into a single long vector. The flatten function below first reads in the value of N 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 H x W x C, but we don't need to specify that explicitly).

NOTE: TensorFlow and PyTorch differ on the default Tensor layout; TensorFlow uses N x H x W x C but PyTorch uses N x C x H x W.

In [16]:
```
def flatten(x):
    """
    Input:
    - TensorFlow Tensor of shape (N, D1, ..., DM)

    Output:
    - TensorFlow Tensor of shape (N, D1 * ... * DM)
    """
    N = tf.shape(x)[0]
    return tf.reshape(x, (N, -1))
```

In [17]:
```
def test_flatten():
    # Clear the current TensorFlow graph.
    tf.reset_default_graph()

    # Stage I: Define the TensorFlow graph describing our computation.
    # In this case the computation is trivial: we just want to flatten
    # a Tensor using the flatten function defined above.

    # Our computation will have a single input, x. We don't know its
    # value yet, so we define a placeholder which will hold the value
    # when the graph is run. We then pass this placeholder Tensor to
    # the flatten function; this gives us a new Tensor which will hold
    # a flattened view of x when the graph is run. The tf.device
    # context manager tells TensorFlow whether to place these Tensors
    # on CPU or GPU.
    with tf.device(device):
        x = tf.placeholder(tf.float32)
        x_flat = flatten(x)

    # At this point we have just built the graph describing our computation,
    # but we haven't actually computed anything yet. If we print x and x_flat
    # we see that they don't hold any data; they are just TensorFlow Tensors
    # representing values that will be computed when the graph is run.
    print('x: ', type(x), x)
    print('x_flat: ', type(x_flat), x_flat)
    print()

    # We need to use a TensorFlow Session object to actually run the graph.
    with tf.Session() as sess:
        # Construct concrete values of the input data x using numpy
        x_np = np.arange(24).reshape((2, 3, 4))
        print('x_np:\n', x_np, '\n')

        # Run our computational graph to compute a concrete output value.
        # The first argument to sess.run tells TensorFlow which Tensor
        # we want it to compute the value of; the feed_dict specifies
        # values to plug into all placeholder nodes in the graph. The
        # resulting value of x_flat is returned from sess.run as a
        # numpy array.
        x_flat_np = sess.run(x_flat, feed_dict={x: x_np})
        print('x_flat_np:\n', x_flat_np, '\n')

        # We can reuse the same graph to perform the same computation
        # with different input data
        x_np = np.arange(12).reshape((2, 3, 2))
        print('x_np:\n', x_np, '\n')
        x_flat_np = sess.run(x_flat, feed_dict={x: x_np})
        print('x_flat_np:\n', x_flat_np)
test_flatten()
```

```
x:  <class 'tensorflow.python.framework.ops.Tensor'> Tensor("Placeholder:0",
dtype=float32, device=/device:CPU:0)
x_flat:  <class 'tensorflow.python.framework.ops.Tensor'> Tensor("Reshape:0",
shape=(?, ?), dtype=float32, device=/device:CPU:0)

x_np:
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

x_flat_np:
 [[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
 [12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23.]]

x_np:
 [[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]

x_flat_np:
 [[ 0.  1.  2.  3.  4.  5.]
 [ 6.  7.  8.  9. 10. 11.]]
```

Barebones TensorFlow: Two-Layer Network

We will now implement our first neural network with TensorFlow: a fully-connected ReLU network with two hidden layers and no biases on the CIFAR10 dataset. For now we will use only low-level TensorFlow operators to define the network; later we will see how to use the higher-level abstractions provided by `tf.keras` to simplify the process.

We will define the forward pass of the network in the function `two_layer_fc` ; this will accept TensorFlow Tensors for the inputs and weights of the network, and return a TensorFlow Tensor for the scores. It's important to keep in mind that calling the `two_layer_fc` function **does not** perform any computation; instead it just sets up the computational graph for the forward computation. To actually run the network we need to enter a TensorFlow Session and feed data to the computational graph.

After defining the network architecture in the `two_layer_fc` function, we will test the implementation by setting up and running a computational graph, feeding zeros to the network and checking the shape of the output.

It's important that you read and understand this implementation.

In [18]:
```
def two_layer_fc(x, params):
    """
    A fully-connected neural network; the architecture is:
    fully-connected layer -> ReLU -> fully connected layer.
    Note that we only need to define the forward pass here; TensorFlow will ta
ke
    care of computing the gradients 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 uni
ts,
    and the output layer will produce scores for C classes.

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

    Returns:
    - scores: A TensorFlow Tensor of shape (N, C) giving classification scores
      for the input data x.
    """
    w1, w2 = params  # Unpack the parameters
    x = flatten(x)   # Flatten the input; now x has shape (N, D)
    h = tf.nn.relu(tf.matmul(x, w1)) # Hidden layer: h has shape (N, H)
    scores = tf.matmul(h, w2)        # Compute scores of shape (N, C)
    return scores
```

In [19]:
```
def two_layer_fc_test():
    # TensorFlow's default computational graph is essentially a hidden global
    # variable. To avoid adding to this default graph when you rerun this cel
l,
    # we clear the default graph before constructing the graph we care about.
    tf.reset_default_graph()
    hidden_layer_size = 42

    # Scoping our computational graph setup code under a tf.device context
    # manager lets us tell TensorFlow where we want these Tensors to be
    # placed.
    with tf.device(device):
        # Set up a placehoder for the input of the network, and constant
        # zero Tensors for the network weights. Here we declare w1 and w2
        # using tf.zeros instead of tf.placeholder as we've seen before - this
        # means that the values of w1 and w2 will be stored in the computation
al
        # graph itself and will persist across multiple runs of the graph; in
        # particular this means that we don't have to pass values for w1 and w
2
        # using a feed_dict when we eventually run the graph.
        x = tf.placeholder(tf.float32)
        w1 = tf.zeros((32 * 32 * 3, hidden_layer_size))
        w2 = tf.zeros((hidden_layer_size, 10))

        # Call our two_layer_fc function to set up the computational
        # graph for the forward pass of the network.
        scores = two_layer_fc(x, [w1, w2])

    # Use numpy to create some concrete data that we will pass to the
    # computational graph for the x placeholder.
    x_np = np.zeros((64, 32, 32, 3))
    with tf.Session() as sess:
        # The calls to tf.zeros above do not actually instantiate the values
        # for w1 and w2; the following line tells TensorFlow to instantiate
        # the values of all Tensors (like w1 and w2) that live in the graph.
        sess.run(tf.global_variables_initializer())

        # Here we actually run the graph, using the feed_dict to pass the
        # value to bind to the placeholder for x; we ask TensorFlow to compute
        # the value of the scores Tensor, which it returns as a numpy array.
        scores_np = sess.run(scores, feed_dict={x: x_np})
        print(scores_np.shape)

two_layer_fc_test()
```

```
(64, 10)
```

Barebones TensorFlow: 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. 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: https://www.tensorflow.org/api_docs/python/tf/nn/conv2d (https://www.tensorflow.org/api_docs/python/tf/nn/conv2d); be careful with padding!

HINT: For biases: https://www.tensorflow.org/performance/xla/broadcasting (https://www.tensorflow.org/performance/xla/broadcasting)

In [20]:
```
def three_layer_convnet(x, params):
    """
    A three-layer convolutional network with the architecture described above.

    Inputs:
    - x: A TensorFlow Tensor of shape (N, H, W, 3) giving a minibatch of image
s
    - params: A list of TensorFlow Tensors giving the weights and biases for t
he
      network; should contain the following:
      - conv_w1: TensorFlow Tensor of shape (KH1, KW1, 3, channel_1) giving
        weights for the first convolutional layer.
      - conv_b1: TensorFlow Tensor of shape (channel_1,) giving biases for the
        first convolutional layer.
      - conv_w2: TensorFlow Tensor of shape (KH2, KW2, channel_1, channel_2)
        giving weights for the second convolutional layer
      - conv_b2: TensorFlow Tensor of shape (channel_2,) giving biases for the
        second convolutional layer.
      - fc_w: TensorFlow Tensor giving weights for the fully-connected layer.
        Can you figure out what the shape should be?
      - fc_b: TensorFlow Tensor giving biases for the fully-connected layer.
        Can you figure out what the shape should be?
    """
    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.
#
    ################################################################################
##
    #layer1
    paddings = tf.constant([[0,0], [2,2], [2,2], [0,0]])
    x = tf.pad(x, paddings, 'CONSTANT')
    conv1 = tf.nn.conv2d(x, conv_w1, strides=[1,1,1,1], padding="VALID")+conv_
b1
    relu1 = tf.nn.relu(conv1)
    #layer2
    paddings = tf.constant([[0,0], [1,1], [1,1], [0,0]])
    x2 = tf.pad(conv1, paddings, 'CONSTANT')
    conv2 = tf.nn.conv2d(x2, conv_w2, strides=[1,1,1,1], padding="VALID")+conv
_b2
    relu2 = tf.nn.relu(conv2)
    #layer3
    relu2 = flatten(relu2)
    scores = tf.matmul(relu2, fc_w) + fc_b
    ################################################################################
##
    #                                END OF YOUR CODE
#
    ################################################################################
##
    return scores
```

After defing the forward pass of the three-layer ConvNet above, run the following cell to test your implementation. Like the two-layer network, we use the `three_layer_convnet` function to set up the computational graph, then run the graph on a batch of zeros just to make sure the function doesn't crash, and produces outputs of the correct shape.

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

In [21]:
```
def three_layer_convnet_test():
    tf.reset_default_graph()

    with tf.device(device):
        x = tf.placeholder(tf.float32)
        conv_w1 = tf.zeros((5, 5, 3, 6))
        conv_b1 = tf.zeros((6,))
        conv_w2 = tf.zeros((3, 3, 6, 9))
        conv_b2 = tf.zeros((9,))
        fc_w = tf.zeros((32 * 32 * 9, 10))
        fc_b = tf.zeros((10,))
        params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
        scores = three_layer_convnet(x, params)

    # Inputs to convolutional layers are 4-dimensional arrays with shape
    # [batch_size, height, width, channels]
    x_np = np.zeros((64, 32, 32, 3))

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        scores_np = sess.run(scores, feed_dict={x: x_np})
        print('scores_np has shape: ', scores_np.shape)

with tf.device('/cpu:0'):
    three_layer_convnet_test()
```

```
scores_np has shape:  (64, 10)
```

Barebones TensorFlow: Training Step

We now define the `training_step` function which sets up the part of the computational graph that performs a single training step. This will take three basic steps:

1. Compute the loss
2. Compute the gradient of the loss with respect to all network weights
3. Make a weight update step using (stochastic) gradient descent.

Note that the step of updating the weights is itself an operation in the computational graph - the calls to `tf.assign_sub` in `training_step` return TensorFlow operations that mutate the weights when they are executed. There is an important bit of subtlety here - when we call `sess.run`, TensorFlow does not execute all operations in the computational graph; it only executes the minimal subset of the graph necessary to compute the outputs that we ask TensorFlow to produce. As a result, naively computing the loss would not cause the weight update operations to execute, since the operations needed to compute the loss do not depend on the output of the weight update. To fix this problem, we insert a **control dependency** into the graph, adding a duplicate `loss` node to the graph that does depend on the outputs of the weight update operations; this is the object that we actually return from the `training_step` function. As a result, asking TensorFlow to evaluate the value of the `loss` returned from `training_step` will also implicitly update the weights of the network using that minibatch of data.

We need to use a few new TensorFlow functions to do all of this:

- For computing the cross-entropy loss we'll use `tf.nn.sparse_softmax_cross_entropy_with_logits`: https://www.tensorflow.org/api_docs/python/tf/nn/sparse_softmax_cross_entropy_with_logits (https://www.tensorflow.org/api_docs/python/tf/nn/sparse_softmax_cross_entropy_with_logits)
- For averaging the loss across a minibatch of data we'll use `tf.reduce_mean`: https://www.tensorflow.org/api_docs/python/tf/reduce_mean (https://www.tensorflow.org/api_docs/python/tf/reduce_mean)
- For computing gradients of the loss with respect to the weights we'll use `tf.gradients`: https://www.tensorflow.org/api_docs/python/tf/gradients (https://www.tensorflow.org/api_docs/python/tf/gradients)
- We'll mutate the weight values stored in a TensorFlow Tensor using `tf.assign_sub`: https://www.tensorflow.org/api_docs/python/tf/assign_sub (https://www.tensorflow.org/api_docs/python/tf/assign_sub)
- We'll add a control dependency to the graph using `tf.control_dependencies`: https://www.tensorflow.org/api_docs/python/tf/control_dependencies (https://www.tensorflow.org/api_docs/python/tf/control_dependencies)

In [22]:
```
def training_step(scores, y, params, learning_rate):
    """
    Set up the part of the computational graph which makes a training step.

    Inputs:
    - scores: TensorFlow Tensor of shape (N, C) giving classification scores f
or
      the model.
    - y: TensorFlow Tensor of shape (N,) giving ground-truth labels for score
s;
      y[i] == c means that c is the correct class for scores[i].
    - params: List of TensorFlow Tensors giving the weights of the model
    - learning_rate: Python scalar giving the learning rate to use for gradien
t
      descent step.

    Returns:
    - loss: A TensorFlow Tensor of shape () (scalar) giving the loss for this
      batch of data; evaluating the loss also performs a gradient descent step
      on params (see above).
    """
    # First compute the loss; the first line gives losses for each example in
    # the minibatch, and the second averages the losses acros the batch
    losses = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=s
cores)
    loss = tf.reduce_mean(losses)

    # Compute the gradient of the loss with respect to each parameter of the t
he
    # network. This is a very magical function call: TensorFlow internally
    # traverses the computational graph starting at loss backward to each elem
ent
    # of params, and uses backpropagation to figure out how to compute gradien
ts;
    # it then adds new operations to the computational graph which compute the
    # requested gradients, and returns a list of TensorFlow Tensors that will
    # contain the requested gradients when evaluated.
    grad_params = tf.gradients(loss, params)

    # Make a gradient descent step on all of the model parameters.
    new_weights = []
    for w, grad_w in zip(params, grad_params):
        new_w = tf.assign_sub(w, learning_rate * grad_w)
        new_weights.append(new_w)

    # Insert a control dependency so that evaluting the loss causes a weight
    # update to happen; see the discussion above.
    with tf.control_dependencies(new_weights):
        return tf.identity(loss)
```

Barebones TensorFlow: Training Loop

Now we set up a basic training loop using low-level TensorFlow operations. We will train the model using stochastic gradient descent without momentum. The `training_step` function sets up the part of the computational graph that performs the training step, and the function `train_part2` iterates through the training data, making training steps on each minibatch, and periodically evaluates accuracy on the validation set.

In [23]:
```
def train_part2(model_fn, init_fn, learning_rate):
    """
    Train a model on CIFAR-10.

    Inputs:
    - model_fn: A Python function that performs the forward pass of the model
      using TensorFlow; it should have the following signature:
      scores = model_fn(x, params) where x is a TensorFlow Tensor giving a
      minibatch of image data, params is a list of TensorFlow Tensors holding
      the model weights, and scores is a TensorFlow Tensor of shape (N, C)
      giving scores for all elements of x.
    - init_fn: A Python function that initializes the parameters of the model.
      It should have the signature params = init_fn() where params is a list
      of TensorFlow Tensors holding the (randomly initialized) weights of the
      model.
    - learning_rate: Python float giving the learning rate to use for SGD.
    """
    # First clear the default graph
    tf.reset_default_graph()
    is_training = tf.placeholder(tf.bool, name='is_training')
    # Set up the computational graph for performing forward and backward passe
s,
    # and weight updates.
    with tf.device(device):
        # Set up placeholders for the data and labels
        x = tf.placeholder(tf.float32, [None, 32, 32, 3])
        y = tf.placeholder(tf.int32, [None])
        params = init_fn()           # Initialize the model parameters
        scores = model_fn(x, params) # Forward pass of the model
        loss = training_step(scores, y, params, learning_rate)

    # Now we actually run the graph many times using the training data
    with tf.Session() as sess:
        # Initialize variables that will live in the graph
        sess.run(tf.global_variables_initializer())
        for t, (x_np, y_np) in enumerate(train_dset):
            # Run the graph on a batch of training data; recall that asking
            # TensorFlow to evaluate loss will cause an SGD step to happen.
            feed_dict = {x: x_np, y: y_np}
            loss_np = sess.run(loss, feed_dict=feed_dict)

            # Periodically print the loss and check accuracy on the val set
            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss_np))
                check_accuracy(sess, val_dset, x, scores, is_training)
```

Barebones TensorFlow: 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. Note that this function accepts a TensorFlow Session object as one of its arguments; this is needed since the function must actually run the computational graph many times on the data that it loads from the dataset `dset`.

Also note that we reuse the same computational graph both for taking training steps and for evaluating the model; however since the `check_accuracy` function never evalutes the `loss` value in the computational graph, the part of the graph that updates the weights of the graph do not execute on the validation data.

In [24]:
```
def check_accuracy(sess, dset, x, scores, is_training=None):
    """
    Check accuracy on a classification model.

    Inputs:
    - sess: A TensorFlow Session that will be used to run the graph
    - dset: A Dataset object on which to check accuracy
    - x: A TensorFlow placeholder Tensor where input images should be fed
    - scores: A TensorFlow Tensor representing the scores output from the
      model; this is the Tensor we will ask TensorFlow to evaluate.

    Returns: Nothing, but prints the accuracy of the model
    """
    num_correct, num_samples = 0, 0
    for x_batch, y_batch in dset:
        feed_dict = {x: x_batch, is_training: 0}
        scores_np = sess.run(scores, feed_dict=feed_dict)
        y_pred = scores_np.argmax(axis=1)
        num_samples += x_batch.shape[0]
        num_correct += (y_pred == y_batch).sum()
    acc = float(num_correct) / num_samples
    print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 * ac
c))
```

Barebones TensorFlow: Initialization

We'll use the following utility method to initialize the weight matrices for our models using Kaiming's normalization method.

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

In [25]:
```
def kaiming_normal(shape):
    if len(shape) == 2:
        fan_in, fan_out = shape[0], shape[1]
    elif len(shape) == 4:
        fan_in, fan_out = np.prod(shape[:3]), shape[3]
    return tf.random_normal(shape) * np.sqrt(2.0 / fan_in)
```

Barebones TensorFlow: Train a Two-Layer Network

We are finally ready to use all of the pieces defined above to train a two-layer fully-connected network on CIFAR-10.

We just need to define a function to initialize the weights of the model, and call `train_part2`.

Defining the weights of the network introduces another important piece of TensorFlow API: `tf.Variable`. A TensorFlow Variable is a Tensor whose value is stored in the graph and persists across runs of the computational graph; however unlike constants defined with `tf.zeros` or `tf.random_normal`, the values of a Variable can be mutated as the graph runs; these mutations will persist across graph runs. Learnable parameters of the network are usually stored in Variables.

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

In [26]:
```
def two_layer_fc_init():
    """
    Initialize the weights of a two-layer network, for use with the
    two_layer_network function defined above.

    Inputs: None

    Returns: A list of:
    - w1: TensorFlow Variable giving the weights for the first layer
    - w2: TensorFlow Variable giving the weights for the second layer
    """
    hidden_layer_size = 4000
    w1 = tf.Variable(kaiming_normal((3 * 32 * 32, 4000)))
    w2 = tf.Variable(kaiming_normal((4000, 10)))
    return [w1, w2]

learning_rate = 1e-2
train_part2(two_layer_fc, two_layer_fc_init, learning_rate)
```

```
Iteration 0, loss = 3.1501
Got 121 / 1000 correct (12.10%)
Iteration 100, loss = 1.8731
Got 389 / 1000 correct (38.90%)
Iteration 200, loss = 1.4257
Got 415 / 1000 correct (41.50%)
Iteration 300, loss = 1.7918
Got 376 / 1000 correct (37.60%)
Iteration 400, loss = 1.6816
Got 431 / 1000 correct (43.10%)
Iteration 500, loss = 1.8785
Got 448 / 1000 correct (44.80%)
Iteration 600, loss = 1.8254
Got 423 / 1000 correct (42.30%)
Iteration 700, loss = 2.0228
Got 446 / 1000 correct (44.60%)
```

Barebones TensorFlow: Train a three-layer ConvNet

We will now use TensorFlow to train a three-layer ConvNet on CIFAR-10.

You need to implement the `three_layer_convnet_init` function. Recall that the architecture of the network is:

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

You don't need to do any hyperparameter tuning, but you should see accuracies above 43% after one epoch of training.

In [27]:
```
def three_layer_convnet_init():
    """
    Initialize the weights of a Three-Layer ConvNet, for use with the
    three_layer_convnet function defined above.

    Inputs: None

    Returns a list containing:
    - conv_w1: TensorFlow Variable giving weights for the first conv layer
    - conv_b1: TensorFlow Variable giving biases for the first conv layer
    - conv_w2: TensorFlow Variable giving weights for the second conv layer
    - conv_b2: TensorFlow Variable giving biases for the second conv layer
    - fc_w: TensorFlow Variable giving weights for the fully-connected layer
    - fc_b: TensorFlow Variable giving biases for the fully-connected layer
    """
    params = None
    ############################################################################
##
    # TODO: Initialize the parameters of the three-layer network.
#
    ############################################################################
##
    conv_w1 = tf.Variable(kaiming_normal([5,5,3,32]))
    conv_b1 = tf.Variable(np.zeros([32]), dtype=tf.float32)
    conv_w2 = tf.Variable(kaiming_normal([3,3,32,16]))
    conv_b2 = tf.Variable(np.zeros([16]), dtype=tf.float32)
    fc_w = tf.Variable(kaiming_normal([32*32*16,10]))
    fc_b = tf.Variable(np.zeros([10]), dtype=tf.float32)
    params = (conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b)
    ############################################################################
##
    #                                END OF YOUR CODE
#
    ############################################################################
##
    return params

learning_rate = 3e-3
train_part2(three_layer_convnet, three_layer_convnet_init, learning_rate)
```

```
Iteration 0, loss = 3.3424
Got 117 / 1000 correct (11.70%)
Iteration 100, loss = 1.6892
Got 380 / 1000 correct (38.00%)
Iteration 200, loss = 1.5116
Got 433 / 1000 correct (43.30%)
Iteration 300, loss = 1.5981
Got 421 / 1000 correct (42.10%)
Iteration 400, loss = 1.5573
Got 480 / 1000 correct (48.00%)
Iteration 500, loss = 1.7381
Got 484 / 1000 correct (48.40%)
Iteration 600, loss = 1.5966
Got 492 / 1000 correct (49.20%)
Iteration 700, loss = 1.6305
Got 516 / 1000 correct (51.60%)
```

Part III: Keras Model API

Implementing a neural network using the low-level TensorFlow API is a good way to understand how TensorFlow works, but it's a little inconvenient - we had to manually keep track of all Tensors holding learnable parameters, and we had to use a control dependency to implement the gradient descent update step. This was fine for a small network, but could quickly become unweildy for a large complex model.

Fortunately TensorFlow provides higher-level packages such as `tf.keras` and `tf.layers` which make it easy to build models out of modular, object-oriented layers; `tf.train` allows you to easily train these models using a variety of different optimization algorithms.

In this part of the notebook we will define neural network models using the `tf.keras.Model` API. To implement your own model, you need to do the following:

1. Define a new class which subclasses `tf.keras.model`. Give your class an intuitive name that describes it, like `TwoLayerFC` or `ThreeLayerConvNet`.
2. In the initializer `__init__()` for your new class, define all the layers you need as class attributes. The `tf.layers` package provides many common neural-network layers, like `tf.layers.Dense` for fully-connected layers and `tf.layers.Conv2D` for convolutional layers. Under the hood, these layers will construct `Variable` Tensors for any learnable parameters. **Warning**: Don't forget to call `super().__init__()` as the first line in your initializer!
3. Implement the `call()` method for your class; this implements the forward pass of your model, and defines the *connectivity* of your network. Layers defined in `__init__()` implement `__call__()` so they can be used as function objects that transform input Tensors into output Tensors. Don't define any new layers in `call()`; any layers you want to use in the forward pass should be defined in `__init__()`.

After you define your `tf.keras.Model` subclass, you can instantiate it and use it like the model functions from Part II.

Module API: Two-Layer Network

Here is a concrete example of using the `tf.keras.Model` API to define a two-layer network. There are a few new bits of API to be aware of here:

We use an `Initializer` object to set up the initial values of the learnable parameters of the layers; in particular `tf.variance_scaling_initializer` gives behavior similar to the Kaiming initialization method we used in Part II. You can read more about it here:
https://www.tensorflow.org/api_docs/python/tf/variance_scaling_initializer
(https://www.tensorflow.org/api_docs/python/tf/variance_scaling_initializer)

We construct `tf.layers.Dense` objects to represent the two fully-connected layers of the model. In addition to multiplying their input by a weight matrix and adding a bias vector, these layer can also apply a nonlinearity for you. For the first layer we specify a ReLU activation function by passing `activation=tf.nn.relu` to the constructor; the second layer does not apply any activation function.

Unfortunately the `flatten` function we defined in Part II is not compatible with the `tf.keras.Model` API; fortunately we can use `tf.layers.flatten` to perform the same operation. The issue with our `flatten` function from Part II has to do with static vs dynamic shapes for Tensors, which is beyond the scope of this

notebook; you can read more about the distinction in the documentation (https://www.tensorflow.org/programmers_guide/faq#tensor_shapes).

In [28]:
```
class TwoLayerFC(tf.keras.Model):
    def __init__(self, hidden_size, num_classes):
        super().__init__()
        initializer = tf.variance_scaling_initializer(scale=2.0)
        self.fc1 = tf.layers.Dense(hidden_size, activation=tf.nn.relu,
                                   kernel_initializer=initializer)
        self.fc2 = tf.layers.Dense(num_classes,
                                   kernel_initializer=initializer)
    def call(self, x, training=None):
        x = tf.layers.flatten(x)
        x = self.fc1(x)
        x = self.fc2(x)
        return x


def test_TwoLayerFC():
    """ A small unit test to exercise the TwoLayerFC model above. """
    tf.reset_default_graph()
    input_size, hidden_size, num_classes = 50, 42, 10

    # As usual in TensorFlow, we first need to define our computational graph.
    # To this end we first construct a TwoLayerFC object, then use it to const
ruct
    # the scores Tensor.
    model = TwoLayerFC(hidden_size, num_classes)
    with tf.device(device):
        x = tf.zeros((64, input_size))
        scores = model(x)

    # Now that our computational graph has been defined we can run the graph
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        scores_np = sess.run(scores)
        print(scores_np.shape)

test_TwoLayerFC()
```

```
W1213 20:39:40.975288 10688 deprecation.py:323] From c:\users\ayhok\appdata\l
ocal\programs\python\python37\lib\site-packages\tensorflow_core\python\autogr
aph\impl\api.py:332: flatten (from tensorflow.python.layers.core) is deprecat
ed and will be removed in a future version.
Instructions for updating:
Use keras.layers.flatten instead.
W1213 20:39:40.976279 10688 deprecation.py:323] From c:\users\ayhok\appdata\l
ocal\programs\python\python37\lib\site-packages\tensorflow_core\python\layers
\core.py:332: Layer.apply (from tensorflow.python.keras.engine.base_layer) is
deprecated and will be removed in a future version.
Instructions for updating:
Please use `layer.__call__` method instead.
W1213 20:39:40.991233 10688 deprecation.py:506] From c:\users\ayhok\appdata\l
ocal\programs\python\python37\lib\site-packages\tensorflow_core\python\ops\re
source_variable_ops.py:1630: calling BaseResourceVariable.__init__ (from tens
orflow.python.ops.resource_variable_ops) with constraint is deprecated and wi
ll be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
```

```
(64, 10)
```

Funtional API: Two-Layer Network

The `tf.layers` package provides two different higher-level APIs for defining neural network models. In the example above we used the **object-oriented API**, where each layer of the neural network is represented as a Python object (like `tf.layers.Dense`). Here we showcase the **functional API**, where each layer is a Python function (like `tf.layers.dense`) which inputs and outputs TensorFlow Tensors, and which internally sets up Tensors in the computational graph to hold any learnable weights.

To construct a network, one needs to pass the input tensor to the first layer, and construct the subsequent layers sequentially. Here's an example of how to construct the same two-layer nework with the functional API.

In [29]:
```
def two_layer_fc_functional(inputs, hidden_size, num_classes):
    initializer = tf.variance_scaling_initializer(scale=2.0)
    flattened_inputs = tf.layers.flatten(inputs)
    fc1_output = tf.layers.dense(flattened_inputs, hidden_size, activation=tf.
nn.relu,
                                 kernel_initializer=initializer)
    scores = tf.layers.dense(fc1_output, num_classes,
                             kernel_initializer=initializer)
    return scores

def test_two_layer_fc_functional():
    """ A small unit test to exercise the TwoLayerFC model above. """
    tf.reset_default_graph()
    input_size, hidden_size, num_classes = 50, 42, 10

    # As usual in TensorFlow, we first need to define our computational graph.
    # To this end we first construct a two layer network graph by calling the
    # two_layer_network() function. This function constructs the computation
    # graph and outputs the score tensor.
    with tf.device(device):
        x = tf.zeros((64, input_size))
        scores = two_layer_fc_functional(x, hidden_size, num_classes)

    # Now that our computational graph has been defined we can run the graph
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        scores_np = sess.run(scores)
        print(scores_np.shape)

test_two_layer_fc_functional()
```

```
W1213 20:39:51.969074 10688 deprecation.py:323] From <ipython-input-29-e2eb30
b3f3fa>:5: dense (from tensorflow.python.layers.core) is deprecated and will
be removed in a future version.
Instructions for updating:
Use keras.layers.Dense instead.

(64, 10)
```

Keras Model API: Three-Layer ConvNet

Now it's your turn to implement a three-layer ConvNet using the `tf.keras.Model` API. Your model should have the same architecture used in Part II:

1. Convolutional layer with 5 x 5 kernels, with zero-padding of 2
2. ReLU nonlinearity
3. Convolutional layer with 3 x 3 kernels, with zero-padding of 1
4. ReLU nonlinearity
5. Fully-connected layer to give class scores

You should initialize the weights of your network using the same initialization method as was used in the two-layer network above.

Hint: Refer to the documentation for `tf.layers.Conv2D` and `tf.layers.Dense`:

https://www.tensorflow.org/api_docs/python/tf/layers/Conv2D (https://www.tensorflow.org/api_docs/python/tf/layers/Conv2D)

https://www.tensorflow.org/api_docs/python/tf/layers/Dense (https://www.tensorflow.org/api_docs/python/tf/layers/Dense)

In [30]:
```
class ThreeLayerConvNet(tf.keras.Model):
    def __init__(self, channel_1, channel_2, num_classes):
        super().__init__()
        ########################################################################
##
        # TODO: Implement the __init__ method for a three-layer ConvNet. You
#
        # should instantiate layer objects to be used in the forward pass.
#
        ########################################################################
##
        init = tf.variance_scaling_initializer(scale=2)
        self.conv1 = tf.layers.Conv2D(channel_1, [5,5], [1,1], padding='valid'
, kernel_initializer= init, activation=tf.nn.relu)
        self.conv2 = tf.layers.Conv2D(channel_2, [3,3], [1,1], padding='valid'
, kernel_initializer= init, activation=tf.nn.relu)
        self.fc = tf.layers.Dense(num_classes, kernel_initializer=init)
        ########################################################################
##
        #                            END OF YOUR CODE
#
        ########################################################################
##

    def call(self, x, training=None):
        scores = None
        ########################################################################
##
        # TODO: Implement the forward pass for a three-layer ConvNet. You
#
        # should use the layer objects defined in the __init__ method.
#
        ########################################################################
##
        x = tf.pad(x, tf.constant([[0,0],[2,2],[2,2],[0,0]]), 'CONSTANT')
        x = self.conv1(x)
        x = tf.pad(x, tf.constant([[0,0],[1,1],[1,1],[0,0]]), 'CONSTANT')
        x = self.conv2(x)
        x = tf.layers.flatten(x)
        scores = self.fc(x)
        ########################################################################
##
        #                            END OF YOUR CODE
#
        ########################################################################
##
        return scores
```

Once you complete the implementation of the `ThreeLayerConvNet` above you can run the following to ensure that your implementation does not crash and produces outputs of the expected shape.

In [31]:
```
def test_ThreeLayerConvNet():
    tf.reset_default_graph()

    channel_1, channel_2, num_classes = 12, 8, 10
    model = ThreeLayerConvNet(channel_1, channel_2, num_classes)
    with tf.device(device):
        x = tf.zeros((64, 3, 32, 32))
        scores = model(x)

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        scores_np = sess.run(scores)
        print(scores_np.shape)

test_ThreeLayerConvNet()
```

```
(64, 10)
```

Keras Model API: Training Loop

We need to implement a slightly different training loop when using the `tf.keras.Model` API. Instead of computing gradients and updating the weights of the model manually, we use an `Optimizer` object from the `tf.train` package which takes care of these details for us. You can read more about `Optimizer` s here: https://www.tensorflow.org/api_docs/python/tf/train/Optimizer (https://www.tensorflow.org/api_docs/python/tf/train/Optimizer)

In [32]:
```
def train_part34(model_init_fn, optimizer_init_fn, num_epochs=1):
    """
    Simple training loop for use with models defined using tf.keras. It trains
    a model for one epoch on the CIFAR-10 training set and periodically checks
    accuracy on the CIFAR-10 validation set.

    Inputs:
    - model_init_fn: A function that takes no parameters; when called it
      constructs the model we want to train: model = model_init_fn()
    - optimizer_init_fn: A function which takes no parameters; when called it
      constructs the Optimizer object we will use to optimize the model:
      optimizer = optimizer_init_fn()
    - num_epochs: The number of epochs to train for

    Returns: Nothing, but prints progress during trainingn
    """
    tf.reset_default_graph()
    with tf.device(device):
        # Construct the computational graph we will use to train the model. We
        # use the model_init_fn to construct the model, declare placeholders f
or
        # the data and labels
        x = tf.placeholder(tf.float32, [None, 32, 32, 3])
        y = tf.placeholder(tf.int32, [None])

        # We need a place holder to explicitly specify if the model is in the
 training
        # phase or not. This is because a number of layers behaves differently
in
        # training and in testing, e.g., dropout and batch normalization.
        # We pass this variable to the computation graph through feed_dict as
 shown below.
        is_training = tf.placeholder(tf.bool, name='is_training')

        # Use the model function to build the forward pass.
        scores = model_init_fn(x, is_training)

        # Compute the loss like we did in Part II
        loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits
=scores)
        loss = tf.reduce_mean(loss)

        # Use the optimizer_fn to construct an Optimizer, then use the optimiz
er
        # to set up the training step. Asking TensorFlow to evaluate the
        # train_op returned by optimizer.minimize(loss) will cause us to make
 a
        # single update step using the current minibatch of data.

        # Note that we use tf.control_dependencies to force the model to run
        # the tf.GraphKeys.UPDATE_OPS at each training step. tf.GraphKeys.UPDA
TE_OPS
        # holds the operators that update the states of the network.
        # For example, the tf.layers.batch_normalization function adds the run
ning mean
        # and variance update operators to tf.GraphKeys.UPDATE_OPS.
```

```
        optimizer = optimizer_init_fn()
        update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
        with tf.control_dependencies(update_ops):
            train_op = optimizer.minimize(loss)

    # Now we can run the computational graph many times to train the model.
    # When we call sess.run we ask it to evaluate train_op, which causes the
    # model to update.
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        t = 0
        for epoch in range(num_epochs):
            print('Starting epoch %d' % epoch)
            for x_np, y_np in train_dset:
                feed_dict = {x: x_np, y: y_np, is_training:1}
                loss_np, _ = sess.run([loss, train_op], feed_dict=feed_dict)
                if t % print_every == 0:
                    print('Iteration %d, loss = %.4f' % (t, loss_np))
                    check_accuracy(sess, val_dset, x, scores, is_training=is_t
raining)
                    print()
                t += 1
```

Keras Model API: Train a Two-Layer Network

We can now use the tools defined above to train a two-layer network on CIFAR-10. We define the `model_init_fn` and `optimizer_init_fn` that construct the model and optimizer respectively when called. Here we want to train the model using stochastic gradient descent with no momentum, so we construct a `tf.train.GradientDescentOptimizer` function; you can [read about it here (https://www.tensorflow.org/api_docs/python/tf/train/GradientDescentOptimizer)](https://www.tensorflow.org/api_docs/python/tf/train/GradientDescentOptimizer).

You don't need to tune any hyperparameters here, but you should achieve accuracies above 40% after one epoch of training.

In [33]:
```
hidden_size, num_classes = 4000, 10
learning_rate = 1e-2

def model_init_fn(inputs, is_training):
    return TwoLayerFC(hidden_size, num_classes)(inputs)

def optimizer_init_fn():
    return tf.train.GradientDescentOptimizer(learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Starting epoch 0
Iteration 0, loss = 3.1910
Got 122 / 1000 correct (12.20%)

Iteration 100, loss = 1.8233
Got 398 / 1000 correct (39.80%)

Iteration 200, loss = 1.4247
Got 407 / 1000 correct (40.70%)

Iteration 300, loss = 1.7992
Got 377 / 1000 correct (37.70%)

Iteration 400, loss = 1.7521
Got 418 / 1000 correct (41.80%)

Iteration 500, loss = 1.8438
Got 428 / 1000 correct (42.80%)

Iteration 600, loss = 1.8526
Got 427 / 1000 correct (42.70%)

Iteration 700, loss = 2.0447
Got 449 / 1000 correct (44.90%)
```

Keras Model API: Train a Two-Layer Network (functional API)

Similarly, we train the two-layer network constructed using the functional API.

In [34]:
```
hidden_size, num_classes = 4000, 10
learning_rate = 1e-2

def model_init_fn(inputs, is_training):
    return two_layer_fc_functional(inputs, hidden_size, num_classes)

def optimizer_init_fn():
    return tf.train.GradientDescentOptimizer(learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Starting epoch 0
Iteration 0, loss = 3.2366
Got 124 / 1000 correct (12.40%)

Iteration 100, loss = 1.8623
Got 393 / 1000 correct (39.30%)

Iteration 200, loss = 1.5939
Got 388 / 1000 correct (38.80%)

Iteration 300, loss = 1.8068
Got 369 / 1000 correct (36.90%)

Iteration 400, loss = 1.8661
Got 421 / 1000 correct (42.10%)

Iteration 500, loss = 1.7380
Got 446 / 1000 correct (44.60%)

Iteration 600, loss = 1.8451
Got 427 / 1000 correct (42.70%)

Iteration 700, loss = 1.9817
Got 460 / 1000 correct (46.00%)
```

Keras Model API: Train a Three-Layer ConvNet

Here you should use the tools we've defined above to train a three-layer ConvNet on CIFAR-10. Your ConvNet should use 32 filters in the first convolutional layer and 16 filters in the second layer.

To train the model you should use gradient descent with Nesterov momentum 0.9.

HINT: https://www.tensorflow.org/api_docs/python/tf/train/MomentumOptimizer (https://www.tensorflow.org/api_docs/python/tf/train/MomentumOptimizer)

You don't need to perform any hyperparameter tuning, but you should achieve accuracies above 45% after training for one epoch.

In [35]:
```
learning_rate = 3e-3
channel_1, channel_2, num_classes = 32, 16, 10

def model_init_fn(inputs, is_training):
    model = None
    ############################################################################
##
    # TODO: Complete the implementation of model_fn.
#
    ############################################################################
##
    model = ThreeLayerConvNet(channel_1, channel_2, num_classes)
    ############################################################################
##
    #                            END OF YOUR CODE
#
    ############################################################################
##
    return model(inputs)

def optimizer_init_fn():
    optimizer = None
    ############################################################################
##
    # TODO: Complete the implementation of model_fn.
#
    ############################################################################
##
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    ############################################################################
##
    #                            END OF YOUR CODE
#
    ############################################################################
##
    return optimizer

train_part34(model_init_fn, optimizer_init_fn)
```

```
Starting epoch 0
Iteration 0, loss = 2.8700
Got 107 / 1000 correct (10.70%)

Iteration 100, loss = 1.8965
Got 358 / 1000 correct (35.80%)

Iteration 200, loss = 1.5417
Got 387 / 1000 correct (38.70%)

Iteration 300, loss = 1.7586
Got 399 / 1000 correct (39.90%)

Iteration 400, loss = 1.7191
Got 426 / 1000 correct (42.60%)

Iteration 500, loss = 1.7794
Got 440 / 1000 correct (44.00%)

Iteration 600, loss = 1.7482
Got 459 / 1000 correct (45.90%)

Iteration 700, loss = 1.7378
Got 457 / 1000 correct (45.70%)
```

Part IV: Keras Sequential API

In Part III we introduced the `tf.keras.Model` API, which allows you to define models with any number of learnable layers and with arbitrary connectivity between layers.

However for many models you don't need such flexibility - a lot of models can be expressed as a sequential stack of layers, with the output of each layer fed to the next layer as input. If your model fits this pattern, then there is an even easier way to define your model: using `tf.keras.Sequential`. You don't need to write any custom classes; you simply call the `tf.keras.Sequential` constructor with a list containing a sequence of layer objects.

One complication with `tf.keras.Sequential` is that you must define the shape of the input to the model by passing a value to the `input_shape` of the first layer in your model.

Keras Sequential API: Two-Layer Network

Here we rewrite the two-layer fully-connected network using `tf.keras.Sequential`, and train it using the training loop defined above.

You don't need to perform any hyperparameter tuning here, but you should see accuracies above 40% after training for one epoch.

In [36]:
```
learning_rate = 1e-2

def model_init_fn(inputs, is_training):
    input_shape = (32, 32, 3)
    hidden_layer_size, num_classes = 4000, 10
    initializer = tf.variance_scaling_initializer(scale=2.0)
    layers = [
        tf.keras.layers.Flatten(input_shape=input_shape),
        tf.keras.layers.Dense(hidden_layer_size, activation=tf.nn.relu,
                              kernel_initializer=initializer),
        tf.keras.layers.Dense(num_classes, kernel_initializer=initializer),
    ]
    model = tf.keras.Sequential(layers)
    return model(inputs)

def optimizer_init_fn():
    return tf.train.GradientDescentOptimizer(learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Starting epoch 0
Iteration 0, loss = 2.7477
Got 118 / 1000 correct (11.80%)

Iteration 100, loss = 1.8811
Got 393 / 1000 correct (39.30%)

Iteration 200, loss = 1.4002
Got 380 / 1000 correct (38.00%)

Iteration 300, loss = 1.6731
Got 374 / 1000 correct (37.40%)

Iteration 400, loss = 1.7971
Got 415 / 1000 correct (41.50%)

Iteration 500, loss = 1.7874
Got 433 / 1000 correct (43.30%)

Iteration 600, loss = 1.8465
Got 427 / 1000 correct (42.70%)

Iteration 700, loss = 1.9846
Got 442 / 1000 correct (44.20%)
```

Keras Sequential API: Three-Layer ConvNet

Here you should use `tf.keras.Sequential` to reimplement the same three-layer ConvNet architecture used in Part II and Part III. As a reminder, your model should have the following architecture:

1. Convolutional layer with 16 5x5 kernels, using zero padding of 2
2. ReLU nonlinearity
3. Convolutional layer with 32 3x3 kernels, using zero padding of 1
4. ReLU nonlinearity
5. Fully-connected layer giving class scores

You should initialize the weights of the model using a `tf.variance_scaling_initializer` as above.

You should train the model using Nesterov momentum 0.9.

You don't need to perform any hyperparameter search, but you should achieve accuracy above 45% after training for one epoch.

In [37]:
```
def model_init_fn(inputs, is_training):
    model = None
    ##############################################################################
    # TODO: Construct a three-layer ConvNet using tf.keras.Sequential.
    #
    ##############################################################################
    inpDim = (32,32,3)
    channel_1 = 32
    channel_2 = 16
    num_classes = 10
    init = tf.variance_scaling_initializer(scale=2)
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.InputLayer(input_shape = inpDim))
    model.add(tf.keras.layers.Conv2D(channel_1, [5,5], [1,1], padding = 'SAME'
,\
                                     kernel_initializer = init,\
                                     activation=tf.nn.relu))
    model.add(tf.keras.layers.Conv2D(channel_2, [5,5], [1,1], padding = 'SAME'
,\
                                     kernel_initializer = init,\
                                     activation=tf.nn.relu))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(num_classes, kernel_initializer = init))
    ##############################################################################
    #                              END OF YOUR CODE                              #
    ##############################################################################
    return model(inputs)

learning_rate = 5e-4
def optimizer_init_fn():
    optimizer = None
    ##############################################################################
    # TODO: Complete the implementation of model_fn.
    #
    ##############################################################################
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9,\
                                           use_nesterov=True)
    ##############################################################################
    #                              END OF YOUR CODE                              #
    ##############################################################################
    return optimizer

train_part34(model_init_fn, optimizer_init_fn)
```

```
Starting epoch 0
Iteration 0, loss = 3.2489
Got 68 / 1000 correct (6.80%)

Iteration 100, loss = 1.7097
Got 378 / 1000 correct (37.80%)

Iteration 200, loss = 1.5194
Got 430 / 1000 correct (43.00%)

Iteration 300, loss = 1.6357
Got 443 / 1000 correct (44.30%)

Iteration 400, loss = 1.4463
Got 459 / 1000 correct (45.90%)

Iteration 500, loss = 1.6524
Got 481 / 1000 correct (48.10%)

Iteration 600, loss = 1.6013
Got 492 / 1000 correct (49.20%)

Iteration 700, loss = 1.5291
Got 502 / 1000 correct (50.20%)
```

Part V: CIFAR-10 open-ended challenge

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

You should experiment with architectures, hyperparameters, loss functions, regularization, or anything else you can think of to train a model that achieves **at least 70%** accuracy on the **validation** set within 10 epochs. You can use the `check_accuracy` and `train` functions from above, or you can implement your own training loop.

Describe what you did at the end of the notebook.

Some things you can try:

- **Filter size**: Above we used 5x5 and 3x3; is this optimal?
- **Number of filters**: Above we used 16 and 32 filters. Would more or fewer do better?
- **Pooling**: We didn't use any pooling above. Would this improve the model?
- **Normalization**: Would your model be improved with batch normalization, layer normalization, group normalization, or some other normalization strategy?
- **Network architecture**: The ConvNet above has only three layers of trainable parameters. Would a deeper model do better?
- **Global average pooling**: Instead of flattening after the final convolutional layer, would global average pooling do better? This strategy is used for example in Google's Inception network and in Residual Networks.
- **Regularization**: Would some kind of regularization improve performance? Maybe weight decay or dropout?

WARNING: Batch Normalization / Dropout

Batch Normalization and Dropout **WILL NOT WORK CORRECTLY** if you use the `train_part34()` function with the object-oriented `tf.keras.Model` or `tf.keras.Sequential` APIs; if you want to use these layers with this training loop then you **must use the tf.layers functional API**.

We wrote `train_part34()` to explicitly demonstrate how TensorFlow works; however there are some subtleties that make it tough to handle the object-oriented batch normalization layer in a simple training loop. In practice both `tf.keras` and `tf` provide higher-level APIs which handle the training loop for you, such as keras.fit (https://keras.io/models/sequential/) and tf.Estimator (https://www.tensorflow.org/programmers_guide/estimators), both of which will properly handle batch normalization when using the object-oriented API.

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 (https://arxiv.org/abs/1512.03385) where the input from the previous layer is added to the output.
  - DenseNets (https://arxiv.org/abs/1608.06993) where inputs into previous layers are concatenated together.
  - This blog has an in-depth overview (https://chatbotslife.com/resnets-highwaynets-and-densenets-oh-my-9bb15918ee32)

Have fun and happy training!

In [ ]:
```
def test_model(model_init_fn):
    tf.reset_default_graph()

    with tf.device(device):
        x = tf.zeros((50, 32, 32, 3))
        scores = model_init_fn(x, True)

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        scores_np = sess.run(scores)
        print(scores_np.shape)


def model_init_fn(inputs, is_training):
    model = None
    ############################################################################
##
    # TODO: Construct a model that performs well on CIFAR-10
#
    ############################################################################
##
    num_classes = 10
    init = tf.variance_scaling_initializer(scale=2.0)
    #conv-relu-dropout-maxpool
    x = tf.layers.conv2d(inputs, 32, [3,3], [1,1], padding = 'SAME', \
                         kernel_initializer=init)
    x = tf.layers.batch_normalization(x, training=is_training)
    x = tf.nn.elu(x)
    x = tf.layers.dropout(x)
    x = tf.layers.max_pooling2d(x, [2,2], [2,2])
    #conv-relu-dropout-maxpool
    x = tf.layers.conv2d(x, 64, [3,3], [1,1], padding = 'VALID',\
                                kernel_initializer=init)
    x = tf.layers.batch_normalization(x, training=is_training)
    x = tf.nn.elu(x)
    x = tf.layers.dropout(x)
    x = tf.layers.max_pooling2d(x, [2,2], [2,2])
    #conv-relu-dropout-maxpool
    x = tf.layers.conv2d(x, 128, [3,3], [1,1], padding = 'VALID',\
                                kernel_initializer=init)
    x = tf.layers.batch_normalization(x, training=is_training)
    x = tf.nn.elu(x)
    x = tf.layers.dropout(x)

    x = tf.layers.average_pooling2d(x, [5,5], [1,1])
    x = tf.layers.flatten(x)

    x = tf.layers.dense(x,50)
    x = tf.layers.dense(x,50)
    net = tf.layers.dense(x,num_classes)
    ############################################################################
##
    #                            END OF YOUR CODE
#
    ############################################################################
##
```

```
    return net

pass

def optimizer_init_fn():
    optimizer = None
    ################################################################################
##
    # TODO: Construct an optimizer that performs well on CIFAR-10
#
    ################################################################################
##
    optimizer = tf.train.AdamOptimizer(learning_rate)
    ################################################################################
##
    #                               END OF YOUR CODE
#
    ################################################################################
##
    return optimizer

device = '/cpu:0'
print_every = 700
num_epochs = 10
train_part34(model_init_fn, optimizer_init_fn, num_epochs)
```

```
W1213 21:14:12.982007 10688 deprecation.py:323] From <ipython-input-38-2bffb4
20e644>:23: conv2d (from tensorflow.python.layers.convolutional) is deprecate
d and will be removed in a future version.
Instructions for updating:
Use `tf.keras.layers.Conv2D` instead.
W1213 21:14:12.998962 10688 deprecation.py:323] From <ipython-input-38-2bffb4
20e644>:24: batch_normalization (from tensorflow.python.layers.normalization)
is deprecated and will be removed in a future version.
Instructions for updating:
Use keras.layers.BatchNormalization instead.  In particular, `tf.control_depe
ndencies(tf.GraphKeys.UPDATE_OPS)` should not be used (consult the `tf.keras.
layers.batch_normalization` documentation).
W1213 21:14:13.044839 10688 deprecation.py:323] From <ipython-input-38-2bffb4
20e644>:26: dropout (from tensorflow.python.layers.core) is deprecated and wi
ll be removed in a future version.
Instructions for updating:
Use keras.layers.dropout instead.
W1213 21:14:13.046834 10688 deprecation.py:323] From <ipython-input-38-2bffb4
20e644>:27: max_pooling2d (from tensorflow.python.layers.pooling) is deprecat
ed and will be removed in a future version.
Instructions for updating:
Use keras.layers.MaxPooling2D instead.
W1213 21:14:13.155569 10688 deprecation.py:323] From <ipython-input-38-2bffb4
20e644>:42: average_pooling2d (from tensorflow.python.layers.pooling) is depr
ecated and will be removed in a future version.
Instructions for updating:
Use keras.layers.AveragePooling2D instead.
```

```
Starting epoch 0
Iteration 0, loss = 2.5543
Got 109 / 1000 correct (10.90%)

Iteration 700, loss = 1.2182
Got 517 / 1000 correct (51.70%)

Starting epoch 1
Iteration 1400, loss = 1.2005
Got 590 / 1000 correct (59.00%)

Starting epoch 2
Iteration 2100, loss = 0.9252
Got 635 / 1000 correct (63.50%)

Starting epoch 3
Iteration 2800, loss = 0.9367
Got 622 / 1000 correct (62.20%)

Starting epoch 4
Iteration 3500, loss = 0.8164
Got 644 / 1000 correct (64.40%)

Starting epoch 5
Iteration 4200, loss = 0.7317
Got 617 / 1000 correct (61.70%)

Starting epoch 6
Iteration 4900, loss = 0.6777
Got 593 / 1000 correct (59.30%)

Starting epoch 7
Iteration 5600, loss = 0.7607
Got 678 / 1000 correct (67.80%)

Starting epoch 8
Iteration 6300, loss = 0.8803
Got 694 / 1000 correct (69.40%)

Starting epoch 9
Iteration 7000, loss = 0.6570
Got 671 / 1000 correct (67.10%)
```

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: Tell us what you did