EEE443 Neural Networks
Homework 3 Report

Name: Oğuz Altan

ID: 21600966

Question 1)

In this question, we implement an autoencoder neural network with a single hidden layer for unsupervised feature extraction from natural images, by minimizing cost function for autoencoder.

Part A)

We are given a dataset of images consisting of 16*16 pixels 10240 images in RGB format. First, we preprocess the data by converting them to grayscale using a luminosity model and do further modifications on the data such as normalization and clipping the data range. 200 random sample patches in RGB format and normalized versions of these sample patches are given below:

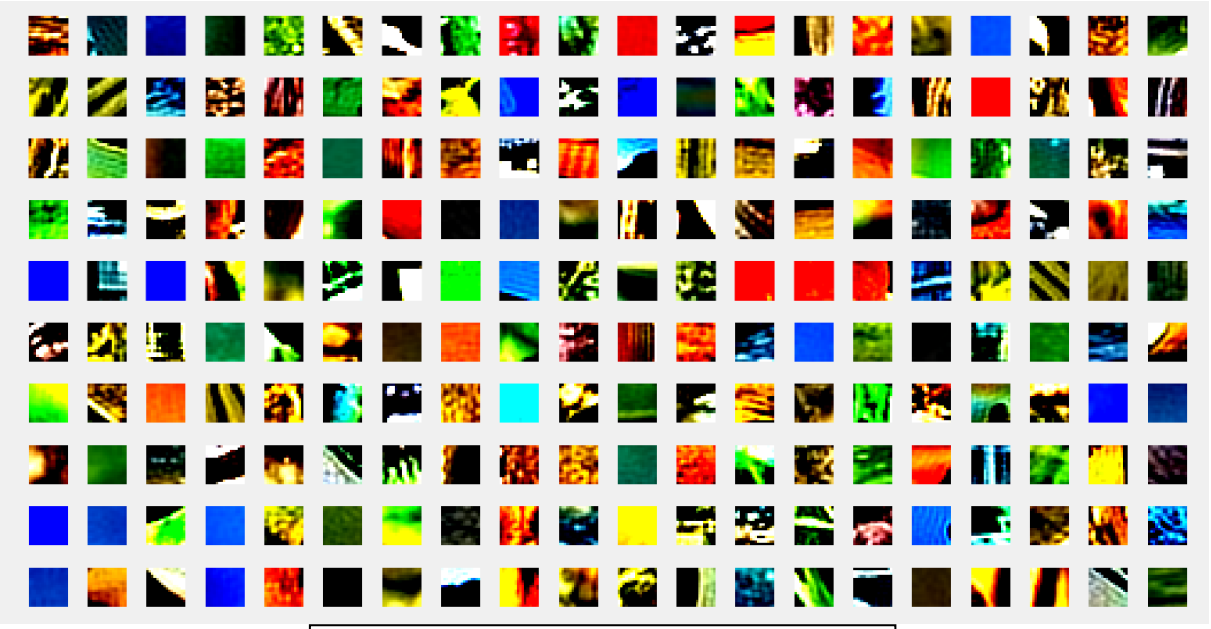


Figure 1: 200 Random Sample Patches in RBG

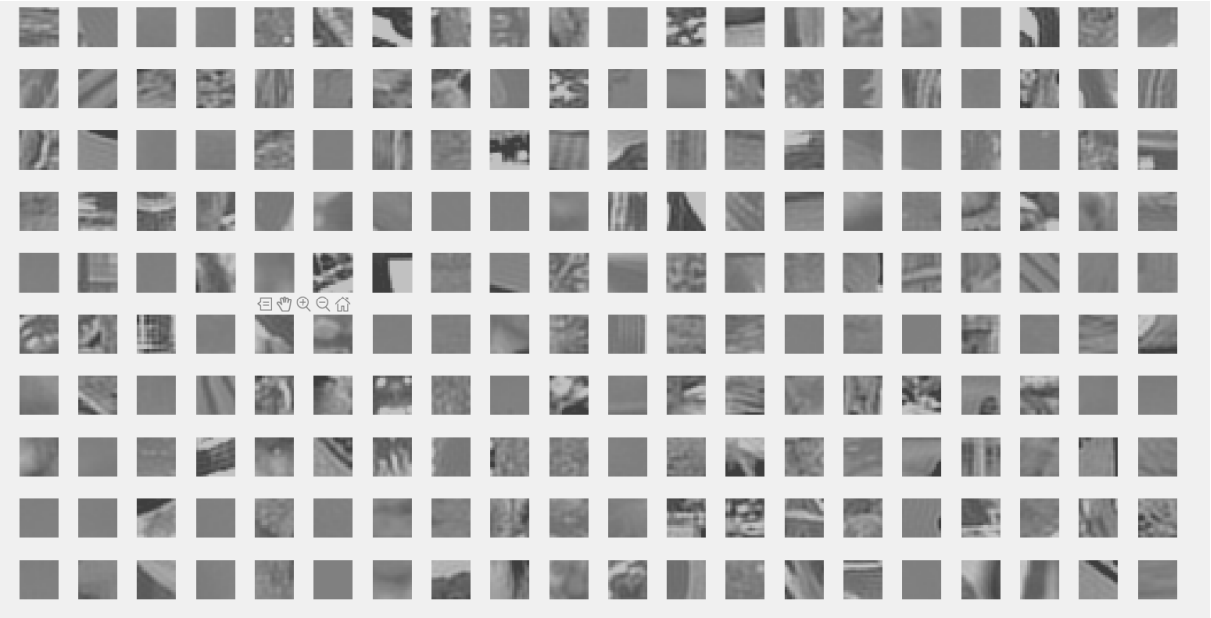


Figure 2: Normalized Versions of 200 Random Sample Patches

Inspecting two figures, it can be seen that normalized versions of the original RBG images are grayscaled versions and the we can detect the different textures and features such as edges and other basic features of original data. Another point is that as normalization brings color loss, monocolor images are almost completely gray, it is difficult to understand the texture on these kind of images.

Part B)

We want to minimize the cost function below:

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

This cost function consists of three separate terms. First term is Mean Squared Error, which is average squared-error between the desired response and the network output across training samples. Second term is regularization term and the third one is Kullback-Leibler divergence term that shows how much the average activation of hidden neurons is different from the sparsity ρ.

KL (Kullback-Leibler) divergence term between a Bernouilli variable with mean ρ and another with mean $\hat{\rho}_b$ can be expressed as below:

$$KL(\rho|\hat{\rho}_b) = \rho\, log_2\left(\frac{\rho}{\hat{\rho}_b}\right) + (1-\rho)\log_2\left(\frac{1-\rho}{1-\hat{\rho}_b}\right)$$

where $\hat{\rho}_b$ is the average of hidden layer neuron activations, calculated through the dataset.

Part C)

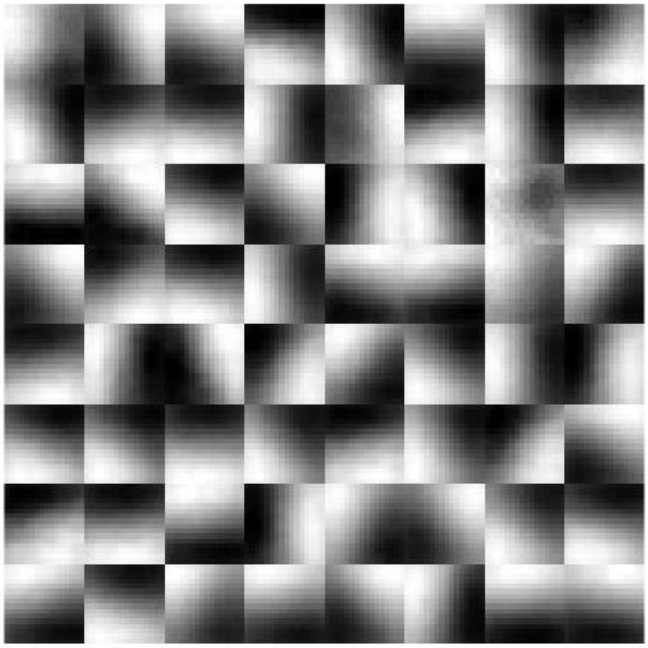


Figure 3: Hidden Layer Weights as a Separate Image for Each Hidden Layer Neuron

The parameters that I found which work well are $\rho = 0.01$ and $\beta = 1.5$. Our hidden layer consists of 64 neurons and first layer of connection weights as a separate image for each neuron in the hidden layer are shown above. These features show the edges and orientations in the corresponding normalized image, therefore the images in the figure above, Figure 3, are decent in representing the original image data.

Part D)

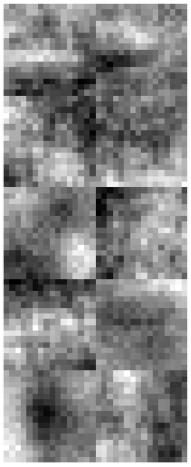


Figure 4: $\lambda = 0$ and L_Hidden = 10

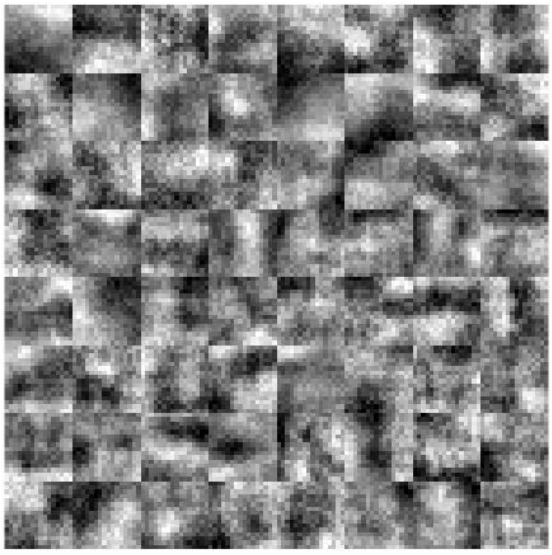


Figure 5: $\lambda = 0$ and L_Hidden = 64

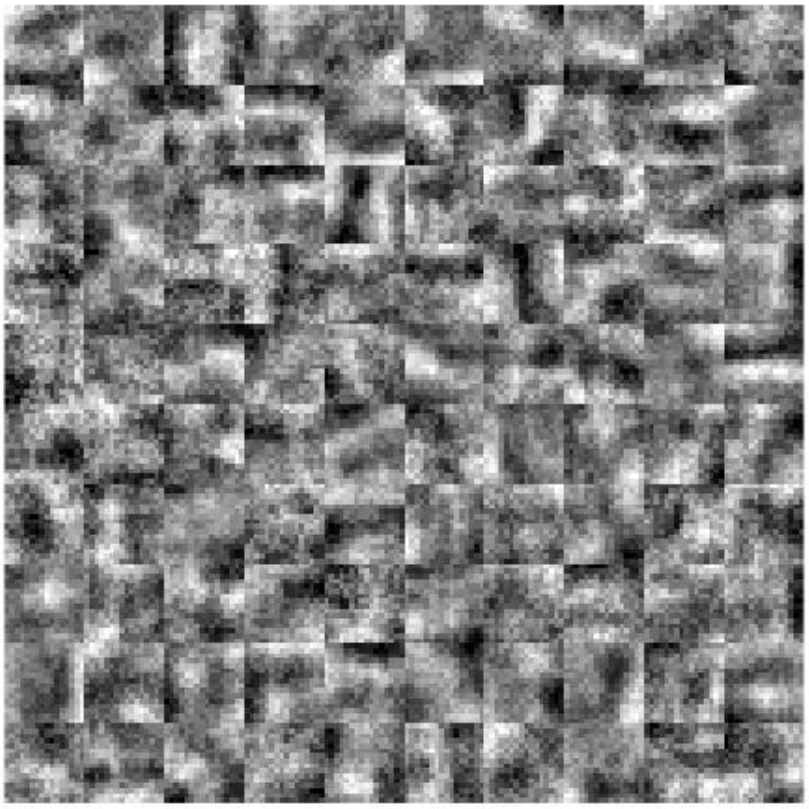


Figure 6: $\lambda = 0$ and L_Hidden = 100

Analyzing the configuration with three different number of hidden layer neurons, denoted as L_Hidden, which are 10, 64 and 100 respectively to the figures, with a constant parameter $\lambda = 0$ we can see the effect of number of hidden layer neurons. As hidden layer neurons learn the features of the original input image data, as the number of hidden layer neurons increase, they well learn the features the original image data, for instance in our case, as the number of hidden layer neurons increase, the network can better detect the edges and orientations in the original images.

Considering the effect of λ, as this parameter is related to the regularization term in the cost function, we can see that happens when λ is 0. Regularization term smoothens out the transitions in the grayscale, so no distinct transitions between white and black, in an grayscale image, if we have an optimal regularization term. For the case of absence of λ, then we do not have any regularization term so we observe sudden, not smooth, transitionts between white and black. This situation can be seen on the figures above.

We increase the λ and train the network again for the same three different number of hidden layer neurons.

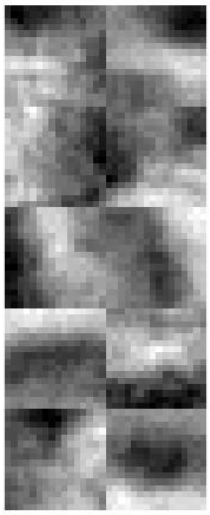


Figure 7: $\lambda = 10^{-4}$ and L_Hidden = 10

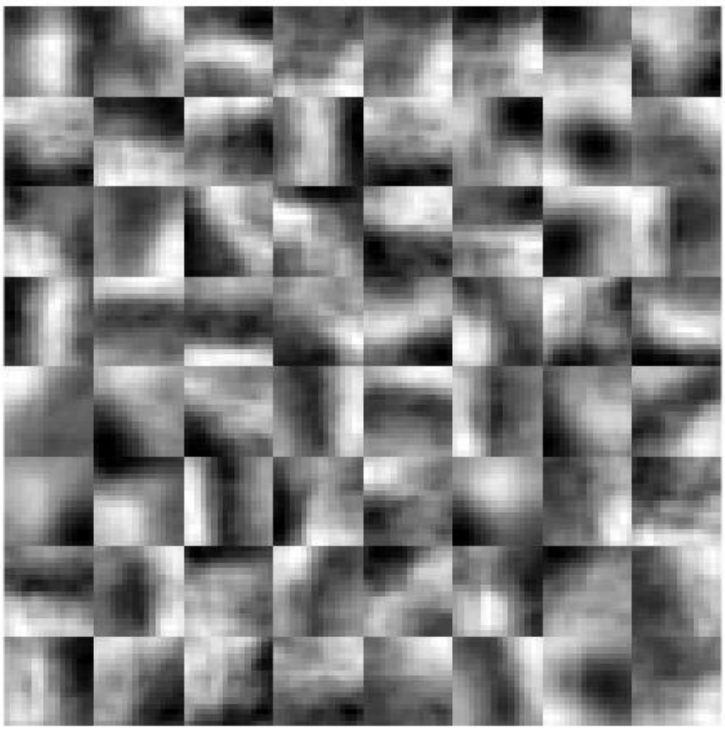


Figure 8: $\lambda = 10^{-4}$ and L_Hidden = 64

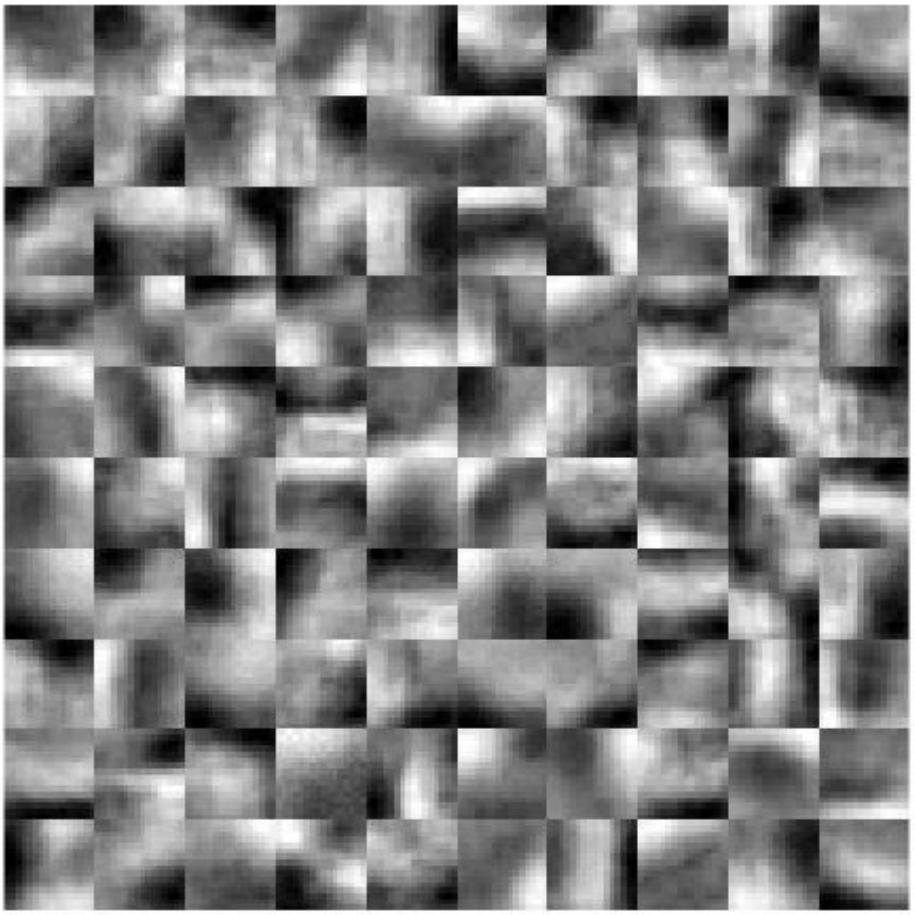


Figure 9: $\lambda = 10^{-4}$ and L_Hidden = 100

The point of regularization term and λ can be seen on the figures above, for the case $\lambda = 10^{-4}$.

The difference between the three figures for the $\lambda = 0$ and $\lambda = 10^{-4}$ is that the edges and orientations on the hidden weight representations for the $\lambda = 10^{-4}$ are much more apparent and transitions are smoother, comparing to the $\lambda = 0$ case. Again, the effect of number of hidden layer neurons is as it is explained above, for the $\lambda = 0$ case.

For the next case, we increase λ to $\lambda = 10^{-3}$ and train the network again for the same three different number of hidden layer neurons.

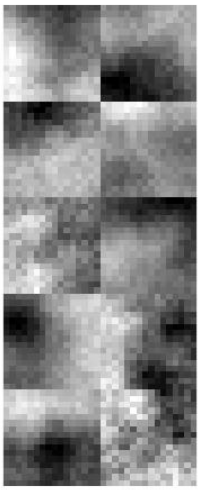


Figure 10: $\lambda = 10^{-3}$ and L_Hidden = 10

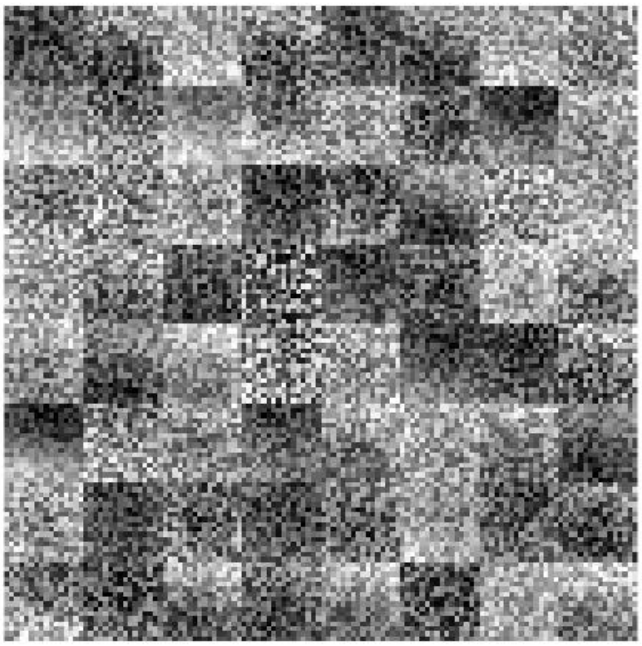


Figure 11: $\lambda = 10^{-3}$ and L_Hidden = 64

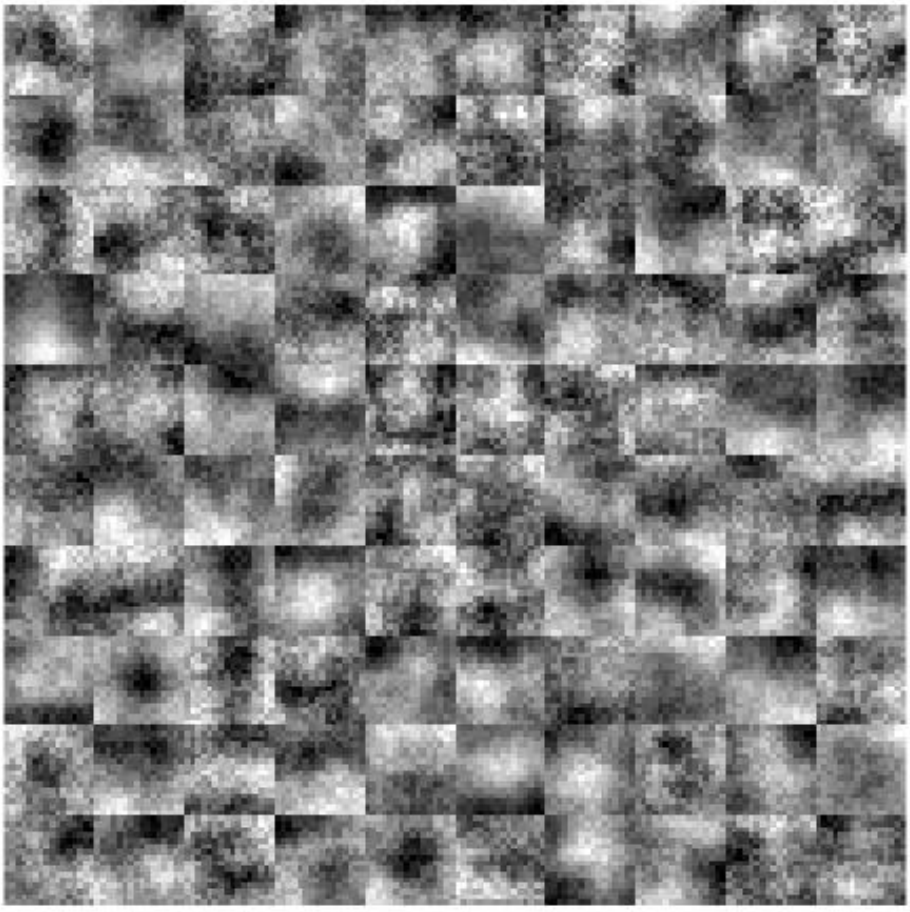


Figure 12: $\lambda = 10^{-3}$ and L_Hidden = 100

As it can be seen on the three figures above, the results are very noisy and transitions are sharp, not smooth. Bigger λ value means that the value of regularization term increases, which results in penaltying the network more harshly. This situation prevents network to learn the features efficiently.

To conclude the analysis on effect of λ and number of hidden layer neurons, as an overall point, as number of hidden layer neurons increase, the more features the network learns of the original image data but increasing the number of hidden layer neurons more than optimal results in overfitting the data, therefore the network becomes worse in predicting the original dataset. For an opposite situation, if the number of hidden layer neurons is small, then the network cannot learn well the features and performs worse. Increasing the λ results in smoother transitions, but if it exceeds the optimal value, the network suffers from noise, which can be seen on the Figure 10, 11 and 12.

Question 2)

Important Note about Comments:

The comments about the outputs and results in this question can be seen just below the every output section in the Jupyter notebook. The results are attached as a PDF to this report.

APPENDIX

```
function oguz_altan_21600966_hw3(question)
clc
close all

switch question
    case '1'
        disp('1')
        %% question 1 code goes here
        clear;
        load assign3_data1;

        %part a
        gray_data = 0.2126*data(:,:,1,:) + 0.7152*data(:,:,2,:) + 0.0722*data(:,:,3,:);
        flat_gray_data = reshape(gray_data,[256, 10240]);
        remove_pix = flat_gray_data - mean(flat_gray_data);

        std_mean = std(remove_pix(:)); %finding std across all pixel in the data
        meanpix = max(min(remove_pix, 3*std_mean), - 3*std_mean) / 3*std_mean;
        normalized_ims = (meanpix + 1) * 0.4 + 0.1;
        norm_in_gray = reshape(normalized_ims,[16,16,10240]); %reshape to matrix 16*16*10240
        rand_patch = randperm(10240);

        figure;
        for i = 1:200
            subplot(10,20,i);
            imshow(data(:,:,:,rand_patch(i)));
        end

        figure;
        for i = 1:200
            subplot(10,20,i);
            imshow(norm_in_gray(:,:,rand_patch(i)))
        end

        %part b
        num_images = 10240; %number of images in the dataset
        params.L_in = 256; %input is 256 pixel image vector

        params.L_hid = 64; %by architecture, number of neurons for output and input are same
        params.lambda = 5e-4;

        params.rho = 0.01;
        params.beta = 1.5;

        options = optimset('MaxIter',250);

        W_interval = sqrt(6/(params.L_in + params.L_hid));
        W_hid = - W_interval + (2 * W_interval) .* rand(params.L_in, params.L_hid);
        W_out = - W_interval + (2 * W_interval) .* rand(params.L_hid, params.L_in);
        b_hid =  rand(1,params.L_hid)*2*W_interval-W_interval;
        b_out =  rand(1,params.L_in)*2*W_interval-W_interval;

        We = [W_hid(:) ; W_out(:) ; b_hid(:) ; b_out(:)];

        costFunction = @(We) aeCost(We,normalized_ims,params);
        [we_opt, cost, ep] = fmincg(costFunction,We,options);

        W1 = reshape(we_opt (1:params.L_hid*params.L_in), params.L_in, params.L_hid);
        show_w(W1);
end
end

function [J, Jgrad] = aeCost(We,data,params)

[~,num_images] = size(data);

%packing up weight and bias matrix
W_hid = reshape(We(1 : params.L_in*params.L_hid),params.L_in,params.L_hid);
W_out = reshape(We(params.L_in*params.L_hid+1 :
params.L_in*params.L_hid*2),params.L_hid,params.L_in);
```

```
b_hid = reshape(We(params.L_in*params.L_hid*2+1 : params.L_in*params.L_hid*2 +
params.L_hid),1,params.L_hid);
b_out = reshape(We(params.L_in*params.L_hid*2 + params.L_hid + 1 : size(We)),1,params.L_in);

%feedforwarding network
hid_act = sigmoid(W_hid'*data - b_hid');
out_nn = sigmoid(W_out'*hid_act - b_out');
rho_hat = mean(hid_act,2);

%calculating errors separated into three terms in the cost function
mse = (1/(2*num_images)).*sum(sum((data-out_nn).^2,2));
regul = (params.lambda/2)*(sum(W_hid.^2,'all') + sum(W_out.^2,'all'));
KL_div = params.beta*sum((params.rho*log2(params.rho./rho_hat) + (1-params.rho)*log2((1-
params.rho)./(1-rho_hat))));
J = mse + regul + KL_div; %calculating total cost
Jgrad_W_hidden = (-(1/num_images)*(((W_out*((data-out_nn).*((1-out_nn).*out_nn))).*((1-
hid_act).*hid_act))*data')+ ...
    params.lambda*W_hid' + params.beta*(1/log(2)).*(-params.rho./rho_hat + (1-params.rho)./(1-
rho_hat)).*(1/num_images).*(((1-hid_act).*hid_act)*data'))';
Jgrad_W_out = (-(1/num_images)*(data-out_nn).*((1-out_nn).*out_nn)*hid_act' +
params.lambda*W_out')';
Jgrad_b_hid = (-(1/num_images)*((W_out*((data-out_nn).*((1-out_nn).*out_nn))).*((1-
hid_act).*hid_act)*(-1*ones(1,num_images))') + ...
    params.beta*(1/log(2)).*(-params.rho./rho_hat + (1-params.rho)./(1-
rho_hat)).*(1/num_images).*(((1-hid_act).*hid_act)*(-1*ones(1,num_images))'))';
Jgrad_b_out = (-(1/num_images)*(data-out_nn).*((1-out_nn).*out_nn)*(-1*ones(1,num_images))')';

Jgrad = [Jgrad_W_hidden(:) ; Jgrad_W_out(:) ; Jgrad_b_hid(:) ; Jgrad_b_out(:)]; %rolling the
matrices to a single vector
end

%this function takes the weights, adjust the contrast and dimensions of the
%images that the weights represent in the corresponding layer neurons.
function [h, array] = show_w (we)

we = we - mean(we(:));
[r,c] = size(we);
weight_height = sqrt(r);
div = divisors(c);
[~, no_of_div] = size(div);
im_height = div(round(no_of_div/2));
im_width = c/im_height;
patch = zeros(im_height*(weight_height), im_height*(weight_height));
weight_counter = 1;

for i = 1 : im_height
    for j = 1 : im_width
        if weight_counter > c
            continue;
        end
        patch((j-1)*(weight_height)+(1:weight_height), (i-
1)*(weight_height)+(1:weight_height)) = reshape(we(:, weight_counter),
weight_height,weight_height) / max(abs(we(:, weight_counter)));
        weight_counter = weight_counter + 1;
    end
end

figure;
we = imagesc(patch,[-1 1]);
colormap(gray);
axis image off
drawnow;

end

function [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)
% Minimize a continuous differentialble multivariate function. Starting point
% is given by "X" (D by 1), and the function named in the string "f", must
% return a function value and a vector of partial derivatives. The Polack-
% Ribiere flavour of conjugate gradients is used to compute search directions,
% and a line search using quadratic and cubic polynomial approximations and the
% Wolfe-Powell stopping criteria is used together with the slope ratio method
% for guessing initial step sizes. Additionally a bunch of checks are made to
% make sure that exploration is taking place and that extrapolation will not
% be unboundedly large. The "length" gives the length of the run: if it is
% positive, it gives the maximum number of line searches, if negative its
```

```
% absolute gives the maximum allowed number of function evaluations. You can
% (optionally) give "length" a second component, which will indicate the
% reduction in function value to be expected in the first line-search (defaults
% to 1.0). The function returns when either its length is up, or if no further
% progress can be made (ie, we are at a minimum, or so close that due to
% numerical problems, we cannot get any closer). If the function terminates
% within a few iterations, it could be an indication that the function value
% and derivatives are not consistent (ie, there may be a bug in the
% implementation of your "f" function). The function returns the found
% solution "X", a vector of function values "fX" indicating the progress made
% and "i" the number of iterations (line searches or function evaluations,
% depending on the sign of "length") used.
%
% Usage: [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)
%
% See also: checkgrad
%
% Copyright (C) 2001 and 2002 by Carl Edward Rasmussen. Date 2002-02-13
%
%
% (C) Copyright 1999, 2000 & 2001, Carl Edward Rasmussen
%
% Permission is granted for anyone to copy, use, or modify these
% programs and accompanying documents for purposes of research or
% education, provided this copyright notice is retained, and note is
% made of any changes that have been made.
%
% These programs and documents are distributed without any warranty,
% express or implied.  As the programs were written for research
% purposes only, they have not been tested to the degree that would be
% advisable in any important application.  All use of these programs is
% entirely at the user's own risk.
%
% [ml-class] Changes Made:
% 1) Function name and argument specifications
% 2) Output display
%

% Read options
if exist('options', 'var') && ~isempty(options) && isfield(options, 'MaxIter')
    length = options.MaxIter;
else
    length = 100;
end


RHO = 0.01;                            % a bunch of constants for line searches
SIG = 0.5;       % RHO and SIG are the constants in the Wolfe-Powell conditions
INT = 0.1;    % don't reevaluate within 0.1 of the limit of the current bracket
EXT = 3.0;                    % extrapolate maximum 3 times the current bracket
MAX = 20;                         % max 20 function evaluations per line search
RATIO = 100;                                      % maximum allowed slope ratio

argstr = ['feval(f, X'];                      % compose string used to call function
for i = 1:(nargin - 3)
    argstr = [argstr, ',P', int2str(i)];
end
argstr = [argstr, ')'];

if max(size(length)) == 2, red=length(2); length=length(1); else red=1; end
S=['Iteration '];

i = 0;                                            % zero the run length counter
ls_failed = 0;                             % no previous line search has failed
fX = [];
[f1 df1] = eval(argstr);                      % get function value and gradient
i = i + (length<0);                                            % count epochs?!
s = -df1;                                        % search direction is steepest
d1 = -s'*s;                                                 % this is the slope
z1 = red/(1-d1);                                  % initial step is red/(|s|+1)

while i < abs(length)                                      % while not finished
    i = i + (length>0);                                      % count iterations?!

    X0 = X; f0 = f1; df0 = df1;                   % make a copy of current values
    X = X + z1*s;                                             % begin line search
```

```
[f2 df2] = eval(argstr);
i = i + (length<0);                                          % count epochs?!
d2 = df2'*s;
f3 = f1; d3 = d1; z3 = -z1;             % initialize point 3 equal to point 1
if length>0, M = MAX; else M = min(MAX, -length-i); end
success = 0; limit = -1;                     % initialize quanteties
while 1
    while ((f2 > f1+z1*RHO*d1) | (d2 > -SIG*d1)) & (M > 0)
        limit = z1;                                         % tighten the bracket
        if f2 > f1
            z2 = z3 - (0.5*d3*z3*z3)/(d3*z3+f2-f3);                 % quadratic fit
        else
            A = 6*(f2-f3)/z3+3*(d2+d3);                                 % cubic fit
            B = 3*(f3-f2)-z3*(d3+2*d2);
            z2 = (sqrt(B*B-A*d2*z3*z3)-B)/A;       % numerical error possible - ok!
        end
        if isnan(z2) | isinf(z2)
            z2 = z3/2;                  % if we had a numerical problem then bisect
        end
        z2 = max(min(z2, INT*z3),(1-INT)*z3);  % don't accept too close to limits
        z1 = z1 + z2;                                           % update the step
        X = X + z2*s;
        [f2 df2] = eval(argstr);
        M = M - 1; i = i + (length<0);                           % count epochs?!
        d2 = df2'*s;
        z3 = z3-z2;                    % z3 is now relative to the location of z2
    end
    if f2 > f1+z1*RHO*d1 | d2 > -SIG*d1
        break;                                                % this is a failure
    elseif d2 > SIG*d1
        success = 1; break;                                             % success
    elseif M == 0
        break;                                                          % failure
    end
    A = 6*(f2-f3)/z3+3*(d2+d3);                      % make cubic extrapolation
    B = 3*(f3-f2)-z3*(d3+2*d2);
    z2 = -d2*z3*z3/(B+sqrt(B*B-A*d2*z3*z3));        % num. error possible - ok!
    if ~isreal(z2) | isnan(z2) | isinf(z2) | z2 < 0   % num prob or wrong sign?
        if limit < -0.5                                  % if we have no upper limit
            z2 = z1 * (EXT-1);                 % the extrapolate the maximum amount
        else
            z2 = (limit-z1)/2;                                  % otherwise bisect
        end
    elseif (limit > -0.5) & (z2+z1 > limit)          % extraplation beyond max?
        z2 = (limit-z1)/2;                                               % bisect
    elseif (limit < -0.5) & (z2+z1 > z1*EXT)       % extrapolation beyond limit
        z2 = z1*(EXT-1.0);                           % set to extrapolation limit
    elseif z2 < -z3*INT
        z2 = -z3*INT;
    elseif (limit > -0.5) & (z2 < (limit-z1)*(1.0-INT))   % too close to limit?
        z2 = (limit-z1)*(1.0-INT);
    end
    f3 = f2; d3 = d2; z3 = -z2;                  % set point 3 equal to point 2
    z1 = z1 + z2; X = X + z2*s;                      % update current estimates
    [f2 df2] = eval(argstr);
    M = M - 1; i = i + (length<0);                           % count epochs?!
    d2 = df2'*s;
end                                                      % end of line search

if success                                           % if line search succeeded
    f1 = f2; fX = [fX' f1]';
    %     fprintf('%s %4i | Cost: %4.6e\r', S, i, f1);
    s = (df2'*df2-df1'*df2)/(df1'*df1)*s - df2;      % Polack-Ribiere direction
    tmp = df1; df1 = df2; df2 = tmp;                         % swap derivatives
    d2 = df1'*s;
    if d2 > 0                                        % new slope must be negative
        s = -df1;                                % otherwise use steepest direction
        d2 = -s'*s;
    end
    z1 = z1 * min(RATIO, d1/(d2-realmin));          % slope ratio but max RATIO
    d1 = d2;
    ls_failed = 0;                                % this line search did not fail
else
    X = X0; f1 = f0; df1 = df0;  % restore point from before failed line search
    if ls_failed | i > abs(length)          % line search failed twice in a row
        break;                             % or we ran out of time, so we give up
```

```
        end
        tmp = df1; df1 = df2; df2 = tmp;                         % swap derivatives
        s = -df1;                                                    % try steepest
        d1 = -s'*s;
        z1 = 1/(1-d1);
        ls_failed = 1;                                    % this line search failed
    end
    %   if exist('OCTAVE_VERSION')
    %     fflush(stdout);
    %   end
    disp("Iteration " + i);
end
%plot(1:length(fX),fX);
% fprintf('\n');
end

function sig = sigmoid(x)
sig = 1 ./ (1+exp(-x));
end
```

Question 2)

Part A - Convolutional Networks with Python

In this part, we experiment with demo of a CNN model in Python using Jupyter notebook with Anaconda.

Convolutional Networks

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

In [2]:
```
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerical_gra
dient
from cs231n.layers import *
from cs231n.fast_layers import *
from cs231n.solver import Solver

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

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

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

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

In [3]:
```
# 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,)
```

The image data are load and dimension are shown. For train dataset, we have 32 * 32 pixel 49000 images in RGB format.

Convolution: Naive forward pass

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

We test our implementation by running the following:

In [5]:
```
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
```

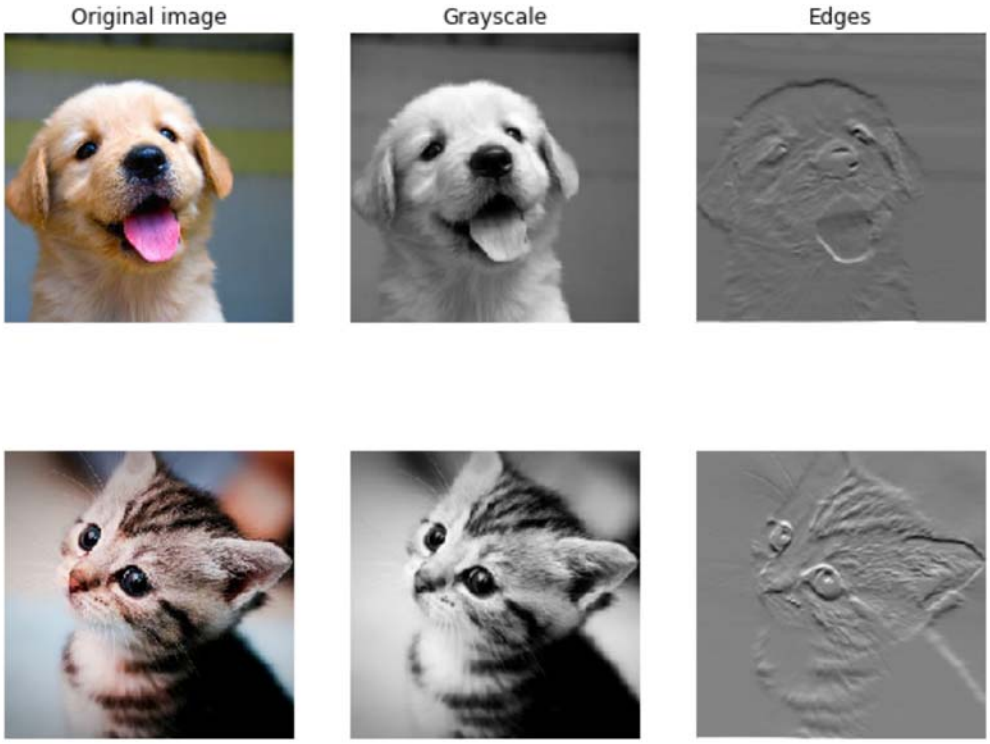
The difference is 2.2121476417505994e-08, which is very close to what is expected, e-8.

Aside: Image processing via convolutions

As fun way to both check our 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 [6]:
```
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()
```

```
D:\Anaconda\lib\site-packages\ipykernel_launcher.py:3: DeprecationWarning: `imre
ad` 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 unti
l
D:\Anaconda\lib\site-packages\ipykernel_launcher.py:10: DeprecationWarning: `imr
esize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # Remove the CWD from sys.path while we load stuff.
D:\Anaconda\lib\site-packages\ipykernel_launcher.py:11: DeprecationWarning: `imr
esize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # This is added back by InteractiveShellApp.init_path()
```



As it can be seen from the 2 example images above, grayscale filter turns the RGB into a grayscale image by modifying the red, green and blue values of the images. Edge detection filter finds the edges of the object in the image.

Convolution: Naive backward pass

We implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cs231n/layers.py`.

We run the following code to check our backward pass with a numeric gradient check.

In [7]:
```
np.random.seed(231)
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b, conv_p
aram)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_p
aram)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_p
aram)[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
```

Our errors for gradients are around e-8, for dx, and even less for other two gradients, which is an expecetesd result.

Max-Pooling: Naive forward

We implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs231n/layers.py`.

We check our implementation by running the following code:

In [8]:
```
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
```

Our difference is 4.1666665157267834e-08, which is on the order of e-8, which is expected.

Max-Pooling: Naive backward

We implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cs231n/layers.py`.

We check our implementation with numeric gradient checking by running the following:

In [9]:
```
np.random.seed(231)
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x, pool_par
am)[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
```

Our error is 3.27562514223145e-12, which is on the order of e-12, which is expected.

Fast layers

We have fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs231n/fast_layers.py` .

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.

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

In [10]:
```
# Rel errors should be around e-9 or less
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(231)
x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

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

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

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

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

```
Testing conv_forward_fast:
Naive: 4.777097s
Fast: 2.742638s
Speedup: 1.741789x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 8.209200s
Fast: 1.418854s
Speedup: 5.785794x
dx difference:  1.949764775345631e-11
dw difference:  5.188375174206562e-13
db difference:  0.0
```

In [11]:
```
# 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: 0.194639s
fast: 0.004491s
speedup: 43.339014x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.480606s
fast: 0.012976s
speedup: 37.037556x
dx difference:  0.0
```

For the part above, the difference is 0.0, which is expected.

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` we will find sandwich layers that implement a few commonly used patterns for convolutional networks.

In [12]:
```
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, co
nv_param, pool_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, co
nv_param, pool_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, co
nv_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:  5.828178746516271e-09
dw error:  8.443628091870788e-09
db error:  3.57960501324485e-10
```

In [13]:
```
from cs231n.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

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

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b, conv_pa
ram)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, conv_pa
ram)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, conv_pa
ram)[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
```

Our dx, dw and db errors are around e-8 or less, which is expected.

Three-layer ConvNet

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

We open the file `cs231n/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class.

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 [14]:
```
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, we use numeric gradient checking to make sure that your backward pass is correct. When we use numeric gradient checking we 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 [15]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
np.random.seed(231)
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

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

The max relative errors for W1, W2, W3, b1, b2 and b3 are very small.

Overfit small data

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

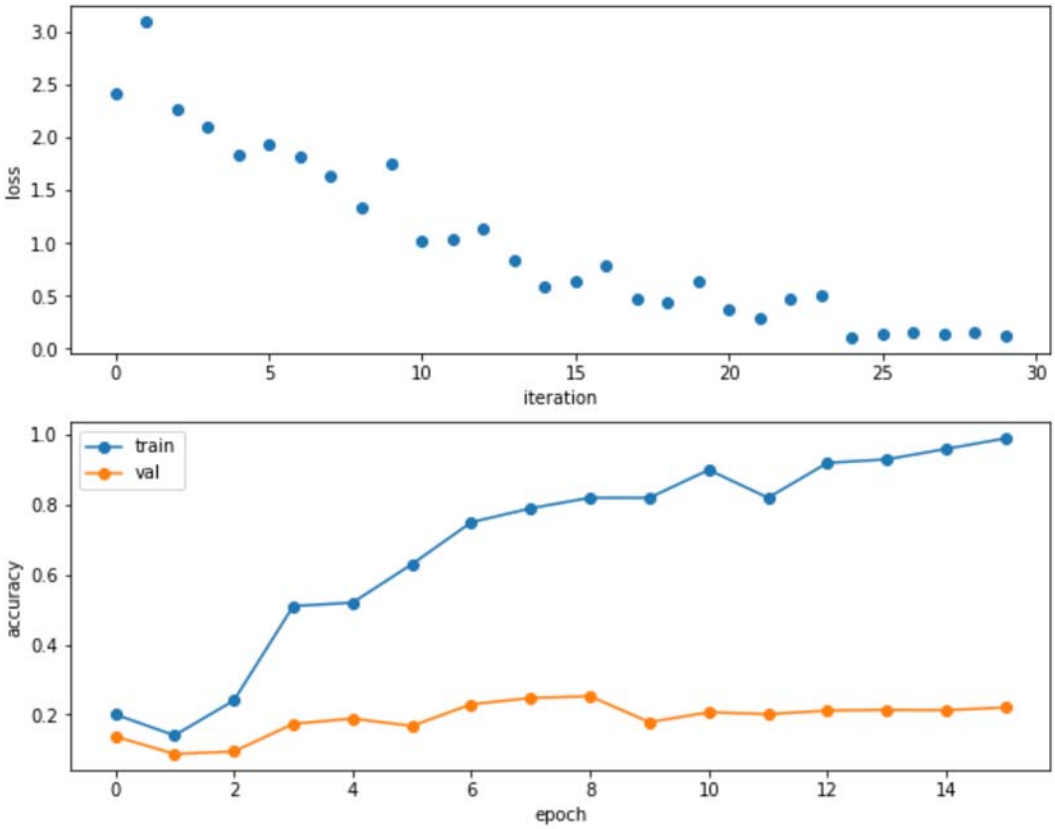
In [16]:
```
np.random.seed(231)

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
```

As it can be seen from the output, the trainin accuracy is very high, about 99%, because we have overfitted a smalldataset in our network.

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

In [17]:
```
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()
```



The overfitting can be clearly seen from the plots above. Especially second plot means a lot, although the accuracy over training dataset increases dramatically, the accuracy over validation dataset stands still, because as the network is overfitted, it learns everything about the training dataset but, in addition, becomes poor at working on new unknown data that it takes as input, validation dataset in our case.

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 [18]:
```
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()
```

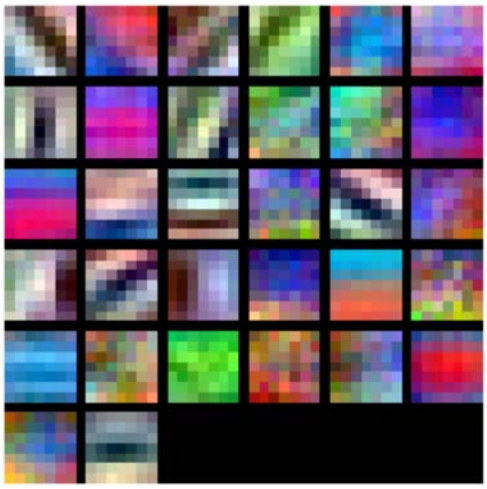
```
(Iteration 1 / 980) loss: 2.304740
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.098229
(Iteration 41 / 980) loss: 1.949788
(Iteration 61 / 980) loss: 1.888398
(Iteration 81 / 980) loss: 1.877093
(Iteration 101 / 980) loss: 1.851877
(Iteration 121 / 980) loss: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

For one epoch, training accuracy is about 50% on the training set.

Visualize Filters

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

In [20]:
```
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`, we implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. We check our 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, 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 [22]:
```
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]
```

After spatial batch, stds become very close to 1 and means very close to 0.

Spatial batch normalization: backward

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

In [23]:
```
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
```

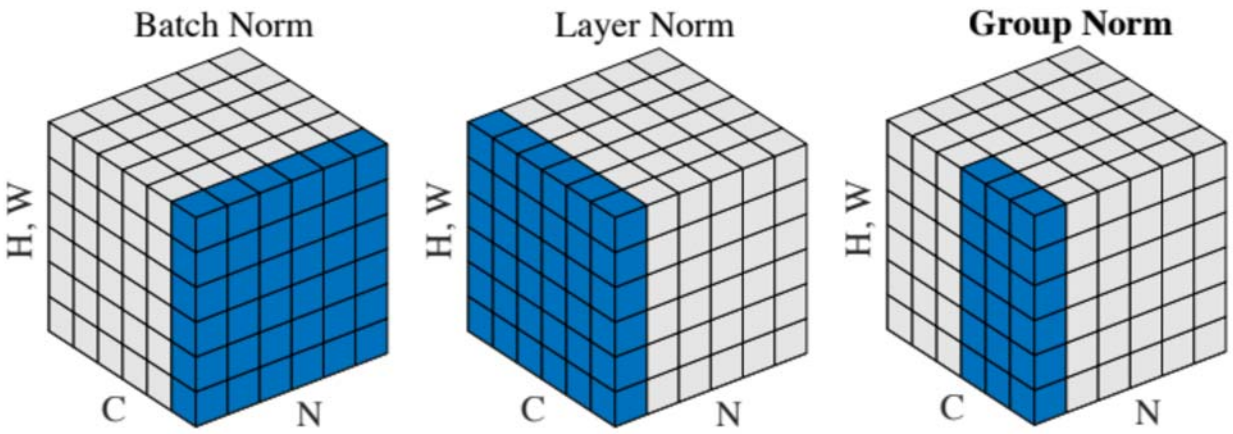
The errors for dx, dgamma and dbeta are between e-12 and e-06, which is expected.

Group Normalization

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

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

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



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

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

We will now implement Group Normalization. Note that this normalization technique that we 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`, we implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. We check our implementation by running the following:

In [25]:
```
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]
```

It can be seen that after spatial group normalization, std are close to one and means close to zero.

Spatial group normalization: backward

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

In [27]:
```
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 [ ]:
```
```

Part B - Convolutional Neural Networks with TensorFlow

In this part, we experiment with demo of a CNN model on TensorFlow library using Jupyter notebook with Anaconda.

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 we should consider using it for our project.

In [1]:
```
import os
import tensorflow as tf
import numpy as np
import math
import timeit
import matplotlib.pyplot as plt
from tensorflow.keras import backend

%matplotlib inline
```

In [2]:
```
def load_cifar10(num_training=49000, num_validation=1000, num_test=10000):
    """
    Fetch the CIFAR-10 dataset from the web and perform preprocessing to prepare
    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,)
```

The dimensions of the dataset and labels are given as output above. Train data consist of 32 - 32 pixel 49000 images in RGB format. Beside, validation and test data set both consist of 32 - 32 pixel 10000 images, again, in RGB format.

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 [3]:
```
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 [4]:
```
# 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,)
```

The output shows how we can form small datasets consist of 64 images, by iterating through a dataset.

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

In [5]:
```
# 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
```

In this assignment, our network runs on CPU.

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 we will specify which parts of the graph we 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 [6]:
```
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 [7]:
```
def test_flatten():
    # Clear the current TensorFlow graph.
    tf.reset_default_graph()
    # tf.compat.v1.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", dty
pe=float32, device=/device:CPU:0)
x_flat:  <class 'tensorflow.python.framework.ops.Tensor'> Tensor("Reshape:0", sh
ape=(?, ?), 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.]]
```

x and x_flat are TensorFlow tensords and they contain placeholders, they will be filled with actual values when the network is run. When we run the network, we obtain x_np and x_flat_np. They are now load with actual data, which means data that we get when we run the network, in numpy format.

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 we read and understand this implementation.

In [8]:
```
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 take
    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 units,
    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 [9]:
```
def two_layer_fc_test():
    # TensorFlow's default computational graph is essentially a hidden global
    # variable. To avoid adding to this default graph when we rerun this cell,
    # 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 computational
        # 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 w2
        # 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)
```

Here, we set up our graph using two_layer_fc_test() function.

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 [46]:
```
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 images
    - params: A list of TensorFlow Tensors giving the weights and biases for the
      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.                #
    ################################################################################

    hid_1 = tf.nn.conv2d(input = x, filter = conv_w1,strides = [1,1,1,1],padding =
'SAME',name = 'conv1') + conv_b1
    hid_11 = tf.nn.relu(hid_1)
    hid_2 = tf.nn.conv2d(input = hid_11,filter = conv_w2,strides = [1,1,1,1],paddin
g = 'SAME' ,name = 'conv2') + conv_b2
    hid_22 = tf.nn.relu(hid_2)
    hid = flatten(hid_22)
    scores = tf.matmul(hid,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 [47]:
```
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)
```

In the code above, we implement feed forward pass algorithm for our network. To check the sanity of the `three_layer_convnet` function, we check the size of the scores_np and it is (64,10), which is expected and correct.

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 [14]:
```
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 for
      the model.
    - y: TensorFlow Tensor of shape (N,) giving ground-truth labels for scores;
      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 gradient
      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=score
s)
    loss = tf.reduce_mean(losses)

    # Compute the gradient of the loss with respect to each parameter of the the
    # network. This is a very magical function call: TensorFlow internally
    # traverses the computational graph starting at loss backward to each element
    # of params, and uses backpropagation to figure out how to compute gradients;
    # 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 [15]:
```
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 passes,
    # 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 [48]:
```
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 * acc))
```

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 [49]:
```
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 [50]:
```
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 = 2.9687
Got 151 / 1000 correct (15.10%)
Iteration 100, loss = 1.9050
Got 381 / 1000 correct (38.10%)
Iteration 200, loss = 1.5824
Got 414 / 1000 correct (41.40%)
Iteration 300, loss = 1.8109
Got 375 / 1000 correct (37.50%)
Iteration 400, loss = 1.8212
Got 428 / 1000 correct (42.80%)
Iteration 500, loss = 1.7243
Got 427 / 1000 correct (42.70%)
Iteration 600, loss = 1.8188
Got 433 / 1000 correct (43.30%)
Iteration 700, loss = 1.8726
Got 432 / 1000 correct (43.20%)
```

When we train our network with two layers, we start with 15.10% correctness percentage at the beginning of the train and after 700 iterations, we get correctness percentage 43.20%.

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 [20]:
```
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.              #
    ############################################################################
    w1 = tf.Variable(kaiming_normal((5,5,3,6)))
    b1 = tf.Variable(kaiming_normal((1,6)))
    w2 = tf.Variable(kaiming_normal((3,3,6,9)))
    b2 = tf.Variable(kaiming_normal((1,9)))
    w = tf.Variable(kaiming_normal((32 * 32 * 9,10)))
    b = tf.Variable(kaiming_normal((1,10)))
    params = [w1,b1,w2,b2,w,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 = 6.3402
Got 81 / 1000 correct (8.10%)
Iteration 100, loss = 1.9967
Got 303 / 1000 correct (30.30%)
Iteration 200, loss = 1.7814
Got 350 / 1000 correct (35.00%)
Iteration 300, loss = 1.8800
Got 345 / 1000 correct (34.50%)
Iteration 400, loss = 1.7001
Got 397 / 1000 correct (39.70%)
Iteration 500, loss = 1.8611
Got 381 / 1000 correct (38.10%)
Iteration 600, loss = 1.7706
Got 434 / 1000 correct (43.40%)
Iteration 700, loss = 1.7182
Got 409 / 1000 correct (40.90%)
```

In the part above, we use Kaiming Initialization to initialize our weights. Instead of using random initializion with normal distribution with mean 0 and variance 1, we use Kaiming Initialization with normal distribution with mean 0 and variance std. Kaiming Initialization helps us to avoid the vanishing gradient problem and exploding gradient problem in the backpropagation phase.

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 [24]:
```
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 construct
    # 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()
```

```
(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 [25]:
```
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.re
lu,
                                 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()
```

```
(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 [26]:
```
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.     #
        ########################################################################
        initializer = tf.variance_scaling_initializer(scale = 2.0)
        self.conv1 = tf.layers.Conv2D(filters = channel_1,kernel_size = [5,5],
                                      strides = [1,1],padding = 'SAME',activation =
tf.nn.relu,
                                     use_bias = True,kernel_initializer = initializ
er,
                                      bias_initializer = initializer,name = 'conv1
')
        self.conv2 = tf.layers.Conv2D(filters = channel_2,kernel_size = [3,3],
                                      strides = [1,1],padding = 'SAME',activation =
tf.nn.relu,
                                     use_bias = True,kernel_initializer = initializ
er,
                                      bias_initializer = initializer,name = 'conv1
')
        self.fc = tf.layers.Dense(units = num_classes,use_bias = True,
                                  kernel_initializer = initializer,bias_initializer
= initializer,
                                  name = 'fc')
        ########################################################################
        #                            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 = self.conv1(x)
        x = self.conv2(x)
        x = tf.layers.flatten(x)
        scores = self.fc(x)
        ########################################################################
        #                            END OF YOUR CODE                          #
        ########################################################################
        return scores
```

We initialize our network parameters and instantiate layer objects with Keras.

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

In [27]:
```
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)
```

We test our `ThreeLayerConvNet`, which initializes the network and instantiate layer objects and see that it gives appropriate output shape which is (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 [28]:
```
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 for
        # 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 train
ing
        # 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=scor
es)
        loss = tf.reduce_mean(loss)

        # Use the optimizer_fn to construct an Optimizer, then use the optimizer
        # 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.UPDATE_OP
S
        # holds the operators that update the states of the network.
        # For example, the tf.layers.batch_normalization function adds the running
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):
```

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).

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

In [29]:
```
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 = 3.1277
Got 129 / 1000 correct (12.90%)

Iteration 100, loss = 1.9017
Got 394 / 1000 correct (39.40%)

Iteration 200, loss = 1.3965
Got 394 / 1000 correct (39.40%)

Iteration 300, loss = 1.9128
Got 384 / 1000 correct (38.40%)

Iteration 400, loss = 1.6867
Got 439 / 1000 correct (43.90%)

Iteration 500, loss = 1.7611
Got 437 / 1000 correct (43.70%)

Iteration 600, loss = 1.8976
Got 428 / 1000 correct (42.80%)

Iteration 700, loss = 1.8702
Got 445 / 1000 correct (44.50%)
```

We train our two layer network using GradientDescentOptimizer function that optimizes the network parameters with stochastic gradient descent with no momentum. At the beginning of the training, the correctness is 12.90% and after 700 iterations, it goes up to 45% accuracy.

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

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

In [30]:
```
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 = 3.0577
Got 158 / 1000 correct (15.80%)

Iteration 100, loss = 1.9497
Got 405 / 1000 correct (40.50%)

Iteration 200, loss = 1.3864
Got 398 / 1000 correct (39.80%)

Iteration 300, loss = 1.7851
Got 392 / 1000 correct (39.20%)

Iteration 400, loss = 1.8592
Got 414 / 1000 correct (41.40%)

Iteration 500, loss = 1.8566
Got 446 / 1000 correct (44.60%)

Iteration 600, loss = 1.8355
Got 428 / 1000 correct (42.80%)

Iteration 700, loss = 1.9461
Got 455 / 1000 correct (45.50%)
```

We follow the same procedure for the training above, namely the partKeras Model API: Train a Two-Layer Network bu this time we use functional API of the Keras. At the beginning of the training, the correctness is 15.80% and after 700 iterations, it goes up to 45.50% accuracy.

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 [31]:
```
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.MomentumOptimizer(learning_rate= 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 = 3.6079
Got 79 / 1000 correct (7.90%)

Iteration 100, loss = 1.6940
Got 404 / 1000 correct (40.40%)

Iteration 200, loss = 1.4717
Got 462 / 1000 correct (46.20%)

Iteration 300, loss = 1.5029
Got 488 / 1000 correct (48.80%)

Iteration 400, loss = 1.3708
Got 494 / 1000 correct (49.40%)

Iteration 500, loss = 1.5152
Got 514 / 1000 correct (51.40%)

Iteration 600, loss = 1.4480
Got 507 / 1000 correct (50.70%)

Iteration 700, loss = 1.4235
Got 513 / 1000 correct (51.30%)
```

This time, we train 3 layer network using Keras API. Our ConvNet uses 32 filters in the first convolutional layer and 16 filters in the second layer. Therefore, we write:

```
model = ThreeLayerConvNet(channel_1,channel_2,num_classes)
```

To set up and initalize our network. To train the modeli we use gradient descent with Nesterov momentum 0.9, which is coded as:

```
optimizer = tf.train.MomentumOptimizer(learning_rate= learning_rate,momentum = 0.9,use
_nesterov = True)
```

When we train out network, we can see that at the beginning of the training, the correctness is 7.90% and after 700 iterations, it goes up to 51.30% accuracy.

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 [35]:
```
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 = 2.9446
Got 150 / 1000 correct (15.00%)

Iteration 100, loss = 1.9076
Got 374 / 1000 correct (37.40%)

Iteration 200, loss = 1.5449
Got 384 / 1000 correct (38.40%)

Iteration 300, loss = 1.8594
Got 372 / 1000 correct (37.20%)

Iteration 400, loss = 1.8357
Got 416 / 1000 correct (41.60%)

Iteration 500, loss = 1.7999
Got 417 / 1000 correct (41.70%)

Iteration 600, loss = 1.9098
Got 404 / 1000 correct (40.40%)

Iteration 700, loss = 2.0334
Got 440 / 1000 correct (44.00%)
```

This time, we use Keras Sequential API, which eases our work as our network is structured of layers that are connected to each other in input-output behaviour. When we train out network, we can see that at the beginning of the training, the correctness is 15.00% and after 700 iterations, it goes up to 44.00% accuracy.

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.           #
    ##############################################################################
    initializer = tf.variance_scaling_initializer(scale = 2.0)
    layers = [
        tf.keras.layers.Conv2D(input_shape = (32,32,3),filters = 16,kernel_size =
[5,5],
                               strides = [1,1],padding = 'SAME',activation = tf.nn.rel
u,
                               use_bias = True,kernel_initializer = initializer,
                               bias_initializer = initializer,name = 'conv1'),
        tf.keras.layers.Conv2D(filters = 32,kernel_size = [5,5],
                               strides = [1,1],padding = 'SAME',activation = tf.nn.rel
u,
                               use_bias = True,kernel_initializer = initializer,
                               bias_initializer = initializer,name = 'conv2'),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(units = 10,use_bias = True,
                               kernel_initializer = initializer,bias_initializer = ini
tializer,
                               name = 'fc')]
    model = tf.keras.Sequential(layers)
    ##############################################################################
    #                               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 = 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 = 3.3606
Got 101 / 1000 correct (10.10%)

Iteration 100, loss = 1.7237
Got 422 / 1000 correct (42.20%)

Iteration 200, loss = 1.3342
Got 466 / 1000 correct (46.60%)

Iteration 300, loss = 1.3113
Got 481 / 1000 correct (48.10%)

Iteration 400, loss = 1.3426
Got 506 / 1000 correct (50.60%)

Iteration 500, loss = 1.5622
Got 506 / 1000 correct (50.60%)

Iteration 600, loss = 1.5478
Got 522 / 1000 correct (52.20%)

Iteration 700, loss = 1.4488
Got 537 / 1000 correct (53.70%)
```

For the three layer network above, we have two convolutional layers, first with 16 5x5 kernels and second with 32 3x3 kernels, which are connected via ReLU activation function. The second convolution layer is fully connected to the output layer that gives class scores. Afte setting up the network, we construct our optimizer, which is momentum optimizer with Nesterov momentum 0.9. When we train out network, we can see that at the beginning of the training, the correctness is 10.10% and after 700 iterations, it goes up to 53.70% accuracy.

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

In [ ]:
```
def model_init_fn(inputs, is_training):
    model = None
    ############################################################################
    # TODO: Construct a model that performs well on CIFAR-10                   #
    ############################################################################
    pass
    ############################################################################
    #                            END OF YOUR CODE                              #
    ############################################################################
    return net

pass

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

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

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