Bilkent University

Department of Electrical & Electronics Engineering

EEE 443 – Neural Networks

Assignment #3

Hasan Emre Erdemoglu

21401462

15/12/2019

TABLE OF CONTENTS

Assignment # 3

Note: In this assignment the preferred coding medium is MATLAB. The entirety of the code can be found in the appendix. Please note that the details in the code implementation is written as comments in the code. Code segments can be also found within the answers to clarify the answers or to show which specific code segment is responsible for generating the output requested by the question.

Question 1:

In this question I will implement a single hidden layer autoencoder using natural image data provided. The cost function which is defined for this problem is given below:

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

Here Kullback-Leibler divergence is described by a Bernoulli variable with mean ρ and another with mean $\widehat{\rho_b}$, which is also described by the average activation of hidden units across training samples.

KL divergence is described by the equation below. It describes how one probability distribution is different than the other one using relative entropy.

$$KL(A|B) = \sum_{i} P(A_i)\ \log\frac{P(A_i)}{P(B_i)}$$

Knowing that knowledge that we have Bernoulli distribution with means ρ and $\widehat{\rho_b}$ we can form the following equation, as we have two discrete classes. Means describe the probability distributions for the Bernoulli case, so I will give the short notation as seen below:

$$KL(\rho||\widehat{\rho_b}) = \rho\log\frac{\rho}{\widehat{\rho_b}} + (1-\rho)\log\frac{(1-\rho)}{(1-\widehat{\rho_b})}$$

$\widehat{\rho_b}$ can be calculated by summing up all activation outputs of the hidden layer and normalizing it by the number of hidden layer units. Knowing these, one may implement the cost function in MATLAB, which will be described in Part B.

Part A:

Part A is trivial, as it just requires pre-processing on the given dataset. The data consists of 10240 images which consists of 16 by 16 images with three color channels. As requested, a luminosity model is applied on each image to convert 3-channel colored images to grayscale images. The following operations are done on the data:

- Pass images from luminosity filter described by:
  - $Y = 0.2126 * R + 0.7152 * G + 0.0722 * B$, where R, G, B are red, green and blue channels respectively.
- As for normalization, respective mean pixel intensity is removed from each image then pixels that are above or below 3 standard deviation range is clipped (Set to ± 3 std range according to its value). For standard deviation clipping, masking is used to speed up calculations.
- 200 random natural image patches are selected, both original and normalized versions are printed in MATLAB.

```
% Preprocess to grayscale w/ model: (Y = 0.2126*R + 0.7152*G + 0.0722*B)
R = squeeze(data(:,:,1,:)); G = squeeze(data(:,:,2,:));
B = squeeze(data(:,:,3,:)); ndata = 0.2126*R + 0.7152*G + 0.0722*B;
clear R; clear G; clear B; % Workspace cleanup

% Calculate mean and std to clip & do normalization
mInts = mean(ndata,3); stdInts = std(ndata,[],3);
ndata = ndata - repmat(mInts,1,1,10240); % mean adjustment
clear mInts; % cleanup

% Do a clip mask and reflect it on dataset:
clip = repmat(3*stdInts, 1 ,1, 10240);
negclip = ndata < -clip; posclip = ndata > clip;
ndata = ndata .* ~negclip + ~negclip .*-clip;
ndata = ndata .* ~posclip + ~posclip .*-clip;

clear stdInts; clear clip;

% Now rescale:
ndata = rescale(ndata, 0.1, 0.9);

% Display random patches:
plotRandomIms(data, ndata);
```

"plotRandomIms" is a separate method which deals with generating subplots for the natural image visualization. It is written this way to ease of read:

```
function plotRandomIms(data, ndata)
disp('Plotting figures, imshow is slow, this will take some time.');
ranidx = randperm(10240, 200);
figure;
for i = 1:200
   subplot(10,20,i);
   imshow(data(:,:,:,ranidx(i)));
end
```

The following figures give the output for Part A of this question:

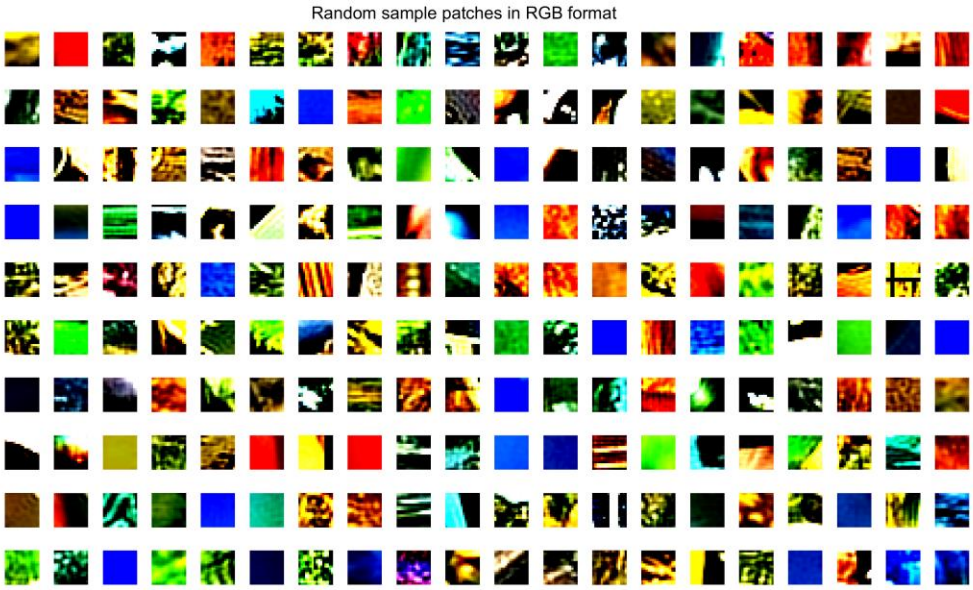


Figure 1. Random sample patches in RGB format

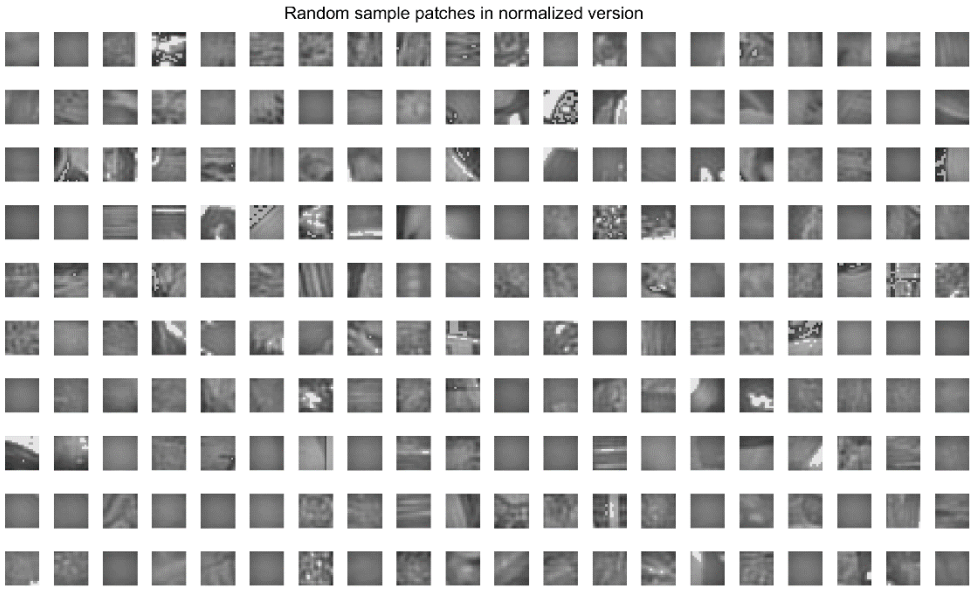


Figure 2. Random sample patches in normalized version

As seen on the natural image patches above, some information retained in the pixels are intensified, whereas some information within the pixels in the image patches are lost. For example, the patches which are green and black are grayed out due to lack of green channel weight in the luminosity model. Due to clipping and scaling some information is not used, hence got lost in the images. Normalization with respect to training samples lowered the contrast between the images.

In general; most of the information are retained after the preprocessing step while avoiding large pixel values that may or may not cause saturations in the activations of the hidden layer. However, images with a single color, most notably green, become mostly indistinguishable to the human eye. This means, there could be a loss in the preprocessing step.

Note that clipping the data will cause a loss in the information retained by the images, however this loss is not very significant given we are clipping at a 3 standard deviation range. For Gaussian distribution such clipping would cause a 0.3 percent loss of information.

Part B:

Part B asks to write an objective function in a specific way and to use a solver to solve for minimum of the objective function. The objective function will calculate the cost which is expressed on top of this question as well as its derivative to make faster calculations of the local/global minimum.

Before starting the derivation and explanation of the code, please note the following remarks:

- I have used mini-batch SGD manually to solve for the minimum, it took a large time to compute the minimum and most of the time failed so I moved to more advanced methods. As the question explicitly states an input to a gradient descent solver, I moved to looking for solvers available in MATLAB.
- I tried using MATLAB's built-in optimizer 'fminunc' which finds minimum of any objective function without any constraints. For Part B, there are 33088 parameters to optimize (two sets of weights, 2 sets of biases). 'fminunc' failed even when gradients are supplied because it must calculate Hessian matrix which is of size 33088 by 33088. By itself this matrix takes 16 GB of memory which freezes the computer. Hessian matrix also needed to be provided, at least partially to avoid such memory complexity. Since 'aeCost' described by Part B does not specify a Hessian calculation, I searched for alternative methods to calculate minimizer function.
- In the internet I have found "fmincg' [1] which calculates the minimum of given objective function when the cost function and its gradient is supplied. The algorithm used in this method is conjugate gradient descent and it does not require a Hessian matrix to calculate the minimum point. The source can be found within the source code, as well as on references of this report.

The objective function 'aeCost' has the following form:

aeCost(we, data, params)

where we is the linearized vector of parameters which have weights from input to hidden layer (w1), weights from hidden layer to output layer (w2), bias from input to hidden layer (b1) and bias

from hidden layer to output layer respectively. ‘params’ is a struct variable which contain number of hidden layers, number of input layers, lambda, beta and rho parameters. Normalized grayscale images are fed to the objective function via ‘data’ parameter. This function calculates the cost defined in the beginning of this question as well as the gradients for the respective weights and biases. The following code describes the objective function.

```
function [J, Jgrad] = aeCost(we,data,params)
% MATLAB doc suggests Jgrad should be given with nargout
N = size(data,2); a1 = data; % Let a1 = data for ease of reading

% Extract data from given parameters:
Lin = params.Lin; Lhid = params.Lhid; lambda = params.lambda;
beta = params.beta; rho = params.rho;

% Unwrap weights for building the autoencoder
[w1,w2,b1,b2] = unwrapper(we, Lin, Lhid);

% Do autoencoding to extract predictions:
[a2, dz2] = sigmoid(w1 * a1 + repmat(b1,1,N));
[a3, dz3] = sigmoid(w2 * a2 + repmat(b2,1,N));

% Calculate rho_b from avg hidden unit activations:
rho_b = mean(a2,2);

% Calculate KL term:
kl = (rho*log(rho./rho_b) + (1-rho)*log((1-rho)./(1-rho_b)));

% Now do the cost operation: - fminunc takes scalar cost (mean)
J = 1/2 * mean(sum(abs(data-a3).^2,1)) + ...
    lambda/2 * (sum(w1(:).^2) + sum(w2(:).^2)) + ...
    beta * sum(kl);

% Derivative calculation: KL term - rho has dependency on rho_b, hence a1
% -- w1,b1 terms Tykhonov term - dependency on w1, w2 terms Avg sq. error
% term - dependency on a2 -- w2,w1,b2,b1 terms Calculate backprop for
% squared term, then add respective partial derivs.

% KL derivative:
dKL = (-rho./rho_b + (1-rho)./(1-rho_b));
dKL = repmat(dKL, 1, N); % repeat for all samples

del3 = -(a1-a3) .* dz3;
dw2 = (del3 * a2') ./ N + lambda * w2;
db2 = sum(del3,2) ./ N;

del2 = (w2' * del3  + beta * dKL).* dz2;
dw1 = (del2 * a1') ./ N + lambda * w1;
db1 = sum(del2,2) ./ N;

if nargout > 1 % gradient required
    % Vectorize all weights:
    Jgrad = [dw1(:); dw2(:); db1(:); db2(:)];
end
end
```

I will go over the code line by line, while constructing the theory behind specific implementations that I listed above.

The code given below reads the data from the parameter to construct the autoencoder. It defines further variables to open the struct variable 'params' into individual variables.

```
N = size(data,2); a1 = data; % Let a1 = data for ease of reading

% Extract data from given parameters:
Lin = params.Lin; Lhid = params.Lhid; lambda = params.lambda;
beta = params.beta; rho = params.rho;

% Unwrap weights for building the autoencoder
[w1,w2,b1,b2] = unwrapper(we, Lin, Lhid);
```

Built-in optimizers only accept a vector of parameters therefore variable we flatten sout all weights and biases and feeds it to the objective function. The method 'unwrapper' is a custom method which divides the weights back to their original shapes. Since weights' and biases' sizes are dependent on input layer and hidden layer sizes, it is easy to divide the vector of hyperparameters. The following is the implementation of this method:

```
function [w1,w2,b1,b2] = unwrapper(we, Lin, Lhid)
% Unwraps weight vector.
w_leng = Lin*Lhid;
w1 = we(1:w_leng); w1 = reshape(w1, Lhid, Lin);
w2 = we(w_leng+1:2*w_leng); w2 = reshape(w2, Lin, Lhid);
b1 = we(2*w_leng+1:2*w_leng+Lhid);
b2 = we(2*w_leng+Lhid+1:end);
end
```

Since w1, w2, b1 and b2 is added tail to tail, it is easy to derive a relationship between when a hyperparameter vector ends and where another one starts. In the case of Part B, input layer size is given as 256 whereas hidden layer size is given as 64. In such case w1 must be of size 64 by 256, b1 must of size 64 by 1, w2 must be of size 256 by 64 and b2 must be of size 256 by 1. Using these entities, it is easy do break down the hyperparameters in the correct form.

```
% Do autoencoding to extract predictions:
[a2, dz2] = sigmoid(w1 * a1 + repmat(b1,1,N));
[a3, dz3] = sigmoid(w2 * a2 + repmat(b2,1,N));

% Calculate rho_b from avg hidden unit activations:
rho_b = mean(a2,2);

% Calculate KL term:
kl = (rho*log(rho./rho_b) + (1-rho)*log((1-rho)./(1-rho_b)));
```

Then it calculates $\widehat{\rho_b}$ as defined in the question by taking the mean value of the hidden layer activations. Finally, it calculates KL divergence term using its definition which was written as:

$$KL(\rho||\widehat{\rho_b}) = \rho \log\frac{\rho}{\widehat{\rho_b}} + (1-\rho)\log\frac{(1-\rho)}{(1-\widehat{\rho_b})}$$

The sigmoid activation is calculated as usual with the following code, this time this function also returns the derivative as well as the sigmoid output. It is done this way, as we need the derivative of the activations when we are doing backpropagation calculations.

```
function [o, do] = sigmoid(z)
% Calculates sigmoid activation for the neurons, z: activation signal
o = 1./(1+exp(-z));
do = o .* (1-o);
end
```

Calculation of the cost function is straight-forward. Cost is defined as:

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

Which is then translated to following code:

```
% Now do the cost operation: - fminunc takes scalar cost (mean)
J = 1/2 * mean(sum(abs(data-a3).^2,1)) + ...
   lambda/2 * (sum(w1(:).^2) + sum(w2(:).^2)) + ...
   beta * sum(kl);
```

Since mean operation do divide by 1/N, I didn't divide by 1/N twice. Entire matrix of weights can be summed by using w1(:) and w2(:) notation as this notation flattens out matrix to a vector directly. Finally, I sum over KL term which I calculated before and sum everything up.

Calculating the derivative of the cost function is trickier, as KL term has dependencies to hidden layer activation. Since the cost consists of three parts which are summed together, it is easy to divide the question into three parts, then using computational graphs the derivatives for respective weights and biases can be calculated by combining derivatives of these parts. The derivative will be calculated in three steps:

1. Derivative of $\frac{1}{2N}\sum_{i=1}^{N}\left|\left|d(i) - o(i)\right|\right|^2$
2. Derivative of $\frac{\lambda}{2}\left[\sum_{b=1}^{L_{hid}}\sum_{a=1}^{L_{in}}\left(W_{a,b}^{(1)}\right)^2 + \sum_{c=1}^{L_{out}}\sum_{a=1}^{L_{hid}}\left(W_{a,c}^{(2)}\right)^2\right]$
3. Derivative of $\beta\sum_{b=1}^{L_{hid}} KL(\rho||\widehat{\rho_b})$

The computation graph significantly simplified differentiation process by graphically showing parameter dependencies. The computational graph for the autoencoder is given below:

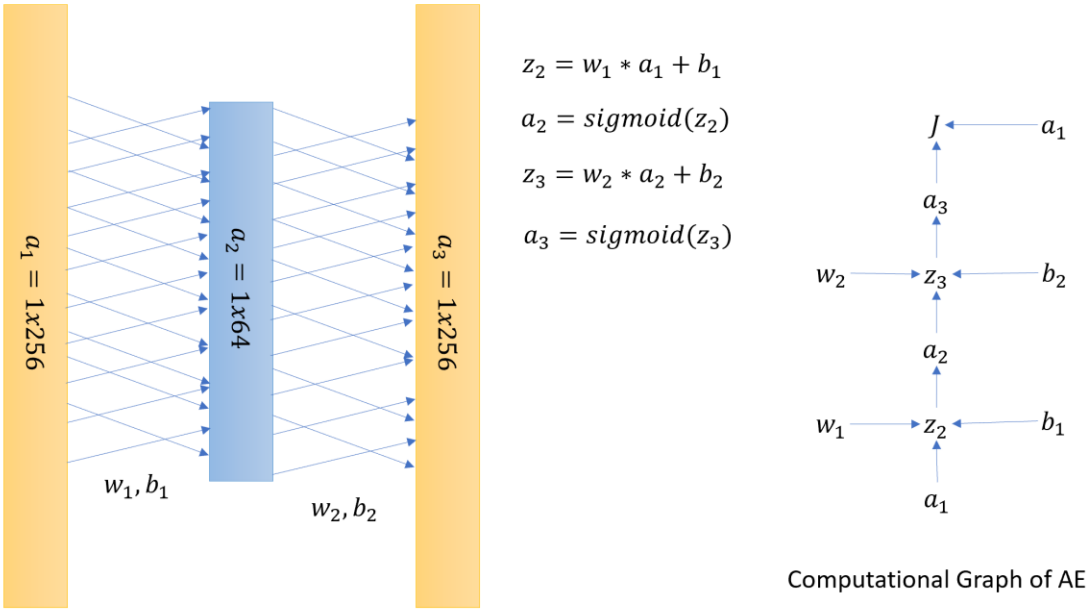


Figure 3. General form and Computational Graph of the Autoencoder

Note that the implementation will tie the weights w1 and w2, where w2 will be equal to the transpose o w1 in initialization.

Derivation of 1:

For this part I will assume that Tikhonov regularization and KL divergence terms do not exist. Since these extra parameters are in sum form their respective gradients can be calculated separately and then they can be added to final gradients for the weights and biases.

The following are the symbolic derivatives under the assumption stated above.

$$\frac{\partial J}{\partial w_2} = \frac{\partial J}{\partial a_3}\frac{\partial a_3}{\partial z_3}\frac{\partial z_3}{\partial w_2}$$

$$\frac{\partial J}{\partial b_2} = \frac{\partial J}{\partial a_3}\frac{\partial a_3}{\partial z_3}\frac{\partial z_3}{\partial b_2}$$

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial a_3}\frac{\partial a_3}{\partial z_3}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial w_1} = \frac{\partial J}{\partial z_3}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial w_1}$$

$$\frac{\partial J}{\partial b_1} = \frac{\partial J}{\partial a_3}\frac{\partial a_3}{\partial z_3}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial b_1} = \frac{\partial J}{\partial z_3}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial b_1}$$

Note that some of the backpropagation calculation from the latter layers are repeated in the preceding layers. While traversing through the computational graph at each layer the previous chain from J to $a_i - z_i$ then $a_i - z_i$ - $a_j - z_j$ gets repeated. This chain effect continues until the start of the neural network. The derivatives for the weights and biases branch out from these chains, therefore storing the computation is vital to improve efficiency of the algorithm.

Using this approach and the computational graph in Figure 3, define the output error as following:

$$\delta_3 = \frac{\partial J}{\partial a_3}\frac{\partial a_3}{\partial z_3} = \frac{\partial J}{\partial a_3} \cdot \varphi'(z_3)$$

Where $\varphi'(z_3)$ calculates the derivative of the sigmoid function given pre-activation z_3. It was noted that while calculating activations of the neurons, I implicitly calculated the derivatives within the sigmoid function for efficiency. Note that in this error term I am doing element-wise multiplication. Now calculating the derivatives hidden layer's weight and bias, I find the following:

$$\frac{\partial J}{\partial w_2} = \delta_3 \frac{\partial z_3}{\partial w_2}\Big/_N = \delta_3 \frac{a_2^T}{N}$$

$$\frac{\partial J}{\partial b_2} = \delta_3 \frac{\partial z_3}{\partial b_2}\Big/_N = \frac{\delta_3}{N}$$

The derivatives $\frac{\partial z_3}{\partial w_2}$ and $\frac{\partial z_3}{\partial b_2}$ come directly from the derivative of the pre-activation function given in Figure 3.As these are matrices, the summation between the training samples occurred implicitly therefore, a division by N is required to normalize the gradients & use transpose operation on the activation function.

In similar way the error for the hidden layer can be written as shown below:

$$\delta_2 = \frac{\partial J}{\partial a_3}\frac{\partial a_3}{\partial z_3}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2} = \delta_3 \frac{\partial z_3}{\partial a_2} \cdot \varphi'(z_2)$$

To fix dimensionality the following notation must be followed:

$$\delta_2 = \left(\frac{\partial z_3}{\partial a_2}^T \delta_3\right) \cdot \varphi'(z_2) = (w_2^T \delta_3) \cdot \varphi'(z_2)$$

Note that also conforms to the notation described with error calculation of the output layer. The only difference is that in output layer, a weighting factor is not in the equation as the output layer is the layer that we stop at.

Now moving on the weights of input layer and bias. Summation over training samples are handled implicitly therefore a division by N is required again.

$$\frac{\partial J}{\partial w_1} = \delta_2 \frac{\frac{\partial z_2}{\partial w_1}}{N} = \delta_2 \frac{a_1^T}{N}$$

$$\frac{\partial J}{\partial b_1} = \delta_2 \frac{\partial z_2}{\partial b_1} \Big/ N = \frac{\delta_2}{N}$$

Note that the relation between derivatives of w_1, w_2 and b_1, b_2 are in the same form. Extending this set of calculations, one may develop a generic backpropagation algorithm for arbitrary number of hidden layers.

I will first do the derivations for the Tikhonov regularization and KL divergence, then show the finalized code I have written to output gradient for the hyperparameters for the cost function.

Derivation of 2:

Derivation of 2 is very straightforward as the sums are only dependent to the weights of the hidden layer and output layer respectively. The part I handle now is below:

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

Since one may move derivative inside the sum the gradients for w_1 and w_2 becomes the following:

$$\frac{\partial J}{\partial w_2} = \lambda \sum_{b=1}^{L_{hid}}\sum_{a=1}^{L_{in}} W_{a,b}^{(1)}$$

$$\frac{\partial J}{\partial w_1} = \lambda \sum_{b=1}^{L_{out}}\sum_{a=1}^{L_{hid}} W_{a,c}^{(2)}$$

These will be added on top of the derivatives I have calculated in section Derivation of 1.

Derivation of 3:

KL divergence has dependencies on the hidden layer neuron activations. Due to this reason it has dependencies to both w_1 and b_1. Since KL divergence term rho, is calculated at the activation layer, its effect would be on the error term of the hidden layer, which in return affects w_1 and b_1. This part of the cost is given below:

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

Continuing the derivation, we achieve:

$$dKL = \beta\left(-\frac{\rho}{\widehat{\rho_b}} + \frac{(1-\rho)}{(1-\widehat{\rho_b})}\right)$$

Merging 1, 2 and 3:

Merging all the parts listed above, the final derivative for the hyperparameters have the forms:

$$\frac{\partial J}{\partial w_2} == \delta_3 \frac{a_2^T}{N} + \lambda \sum_{b=1}^{L_{out}} \sum_{a=1}^{L_{hid}} W_{a,c}^{(2)} = (a_3 - a_1)\frac{a_2^T}{N} + \lambda \sum_{b=1}^{L_{out}} \sum_{a=1}^{L_{hid}} W_{a,c}^{(2)}$$

$$\frac{\partial J}{\partial b_2} = \delta_3 \frac{\partial z_3}{\partial b_2} \Big/ N = \frac{\delta_3}{N} = \frac{(a_3 - a_1)}{N}$$

$$\delta_2 = (w_2^T \delta_3 + \beta dKL) \cdot \varphi'(z_2)$$

$$\frac{\partial J}{\partial w_1} = \delta_2 \frac{a_1^T}{N} + \lambda \sum_{b=1}^{L_{hid}} \sum_{a=1}^{L_{in}} W_{a,b}^{(1)}$$

$$\frac{\partial J}{\partial b_1} = \frac{\delta_2}{N}$$

After doing these operations, the MATLAB code realizing this logic is seen below:

```
% Derivative calculation: KL term - rho has dependency on rho_b, hence a1
% -- w1,b1 terms Tykhonov term - dependency on w1, w2 terms Avg sq. error
% term - dependency on a2 -- w2,w1,b2,b1 terms Calculate backprop for
% squared term, then add respective partial derivs.

% KL derivative:
dKL = (-rho./rho_b + (1-rho)./(1-rho_b));
dKL = repmat(dKL, 1, N); % repeat for all samples

del3 = -(a1-a3) .* dz3;
dw2 = (del3 * a2') ./ N + lambda * w2;
db2 = sum(del3,2) ./ N;

del2 = (w2' * del3  + beta * dKL).* dz2;
dw1 = (del2 * a1') ./ N + lambda * w1;
db1 = sum(del2,2) ./ N;
```

Finally, built-in optimizers of MATLAB require calculated gradient as a variable parameter. Therefore, the gradient is returned to the function handle by using the following code:

```
if nargout > 1 % gradient required
   % Vectorize all weights:
   Jgrad = [dw1(:); dw2(:); db1(:); db2(:)];
end
```

If function output does not have Jgrad, MATLAB does not return the gradient as shown with the code segment above.

After defining the objective function, Part B requires me to experiment upon various beta and rho parameters while keeping lambda and number of hidden units fixed. The following code generate a set of structs called 'params' which is used in defining cost functions in later stages of Part B. Since part D also have a similar experiment setup, this code also has a choice parameter which allows implementing different experiments.

```
function [params] = generateParamSet(Lhids, lambdas, betas, rhos, choice)
if choice == 1
   % For Part B
   c = 1;
   for i = 1: length(betas)
      for j = 1: length(rhos)
         params(c) = struct('Lin', 256, 'Lhid', Lhids, 'lambda',5e-4, ...
            'beta', betas(i), 'rho', rhos(j));
         c = c+1;
      end
   end
else
   % For Part D:
   c = 1;
   for i = 1: length(Lhids)
      for j = 1: length(lambdas)
         params(c) = struct('Lin', 256, 'Lhid', Lhids(i), 'lambda', ...
            lambdas(j),'beta', betas, 'rho', rhos);
         c = c+1;
      end
   end
end
end
```

Rho can be between 0 and 1, where beta can take any arbitrary value. For experimenting upon various parameters, I first have done a search with a large range of values.

```
betas = linspace(0.001,1000,10);
rhos = linspace(0.01,0.99,5);
```

Figure 4. First Experiment Large Beta Space

```
Doing experiments, please wait ... 50/50.
Best cost found in index: 2, w/ cost: 0.65873.
Best beta value is: 0.001, w/ index 1.
Best rho value is: 0.255, w/ index 2.
Note that Jmap can be used to do further refinements on params.
```

Figure 5. Results of First Experiment

As I noticed the cost function is minimized in (beta, rho) = (0.001, 0.255), I have narrowed down my range around these points and ran another 50 experiments to understand 2D hypersurface of the cost.

```
betas = linspace(0.0001,0.01,10);
rhos = linspace(0.1,0.3,5);
```

Figure 6. Range of values in second experiment

```
Doing experiments, please wait ... 50/50.
Cost value = 0.54909.
Best cost found in index: 4, w/ cost: 0.51478.
Best beta value is: 0.0001, w/ index 1.
Best rho value is: 0.25, w/ index 4.
Note that Jmap can be used to do further refinements on params.
```

Figure 7. Results of Second Experiment

One may continue smaller regions to exactly pinpoint the local optima or use larger sampling size extract more points from the cost hyperspace, such implementations are computationally heavy, therefore I have stopped at this point.

Since this is an unsupervised approach, the minimization of the cost normalized squared loss reflects better encoding in the hidden layer, Therefore checking the cost at particular (beta, rho) locations and picking the minimum cost will yield us with a set of parameters that work well for this question.

To experiment on the set of parameters rho and beta, the following code was written. This code is used to output the results given in Figures from 4 to 7. As described in Part B, this code first creates a randomized set of hyperparameters using uniform Xavier initialization. Then, it generates objective function given the data and the parameters initialized before. Finally, for each beta and rho pair being tested it uses custom function “fmincg” to search for the local minima of the objective function with conjugate gradient descent method, using the cost and derivatives provided to the function.

```
function [weStar, bestBeta, bestRho, betastar, rhost] = ...
    partB_experiments(params, ndata, betas, rhos)
J_min = inf; J_min_idx = -1;  % for min test
for exps = 1: length(params)

    % The weights and biases are symm. in 1 layer case so this is enough:
    wo = sqrt(6/(params(exps).Lhid+params(exps).Lin));

    w1 = -wo + rand(params(exps).Lhid,params(exps).Lin) * 2 * wo;
    w2 = w1'; % tied weights
    b1 = -wo + rand(params(exps).Lhid,1) * 2 * wo;
    b2 = -wo + rand(params(exps).Lin,1) * 2 * wo;

    we = [w1(:); w2(:); b1; b2];

    % Convert everything to double
    we = double(we);

    disp(['Doing experiments, please wait ... ', num2str(exps), '/', ...
        num2str(length(params)), '.']) ;

    % USE fminunc to do gradient descent: - minimum unconstrainted
    opts = optimset('GradObj','on','MaxIter',100); % used in fmincg
```

```
    fcn = @(w)aeCost(w,ndata,params(exps));
    [weFinal] = fmincg(fcn, we, opts); % Conj.Gradient Descent
    % I asked this to the course asistant, I was not able to use fminunc as
    % hessian matrix is of 33088 by 33088 size. Question asks for a solver
    % hence I used this one.

    % Now evaluate to see the cost and store it, always keep the minimum
    % cost:
    [J(exps),~] = feval(fcn,weFinal);
    disp(['Cost value = ' , num2str(J(exps)), '.']);
    if J_min > J(exps)
        J_min = J(exps);
        J_min_idx = exps;
        weStar = weFinal;
    end

end

% Find best beta, rho:
betastar = ceil(J_min_idx / length(rhos));
if mod(J_min_idx,length(rhos)) == 0
    rhost = length(rhos);
else
    rhost = mod(J_min_idx,length(rhos));
end

Jmap = reshape(J, length(betas), length(rhos));

% Display best results:
disp(['Best cost found in index: ', num2str(J_min_idx), ', w/ cost: ', ...
    num2str(J_min), '.']);
disp(['Best beta value is: ', num2str(betas(betastar)), ...
    ', w/ index ' , num2str(betastar) , '.']);
disp(['Best rho value is: ', num2str(rhos(rhost)), ', w/ index ' , ...
    num2str(rhost) , '.']);

bestBeta = betas(betastar);
bestRho = rhos(rhost);

figure; imagesc(Jmap); title(' 2D Experiment Cost Values (Jmap)');
xlabel('rho indices'), ylabel('beta indices'); colorbar;
disp('Note that Jmap can be used to do further refinements on params.');

% ************************************************************************
% NOTE: The solver used in this question is appended below. It is taken
% from the link provided below: (broken into two lines, modified)
% https://www.mathworks.com/matlabcentral/mlc-downloads/downloads/
% submissions/56393/versions/1/previews/mathwork/fmincg.m/index.html
% ************************************************************************
end
```

The working specifics of “fmincg” will not be explained in this report. It is a conjugate gradient algorithm which is not readily built into MATLAB. Since the question asks us to use a solver, and built-in methods of MATLAB cannot handle very large number of hyperparameters when calculating the Hessians, due to memory constraints: I had no chance to add and use this method in my assignment. The link where I have found this method is listed in the source code and can be found in the comments above.

After doing the experiments, the code provides above also checks for the params and the hyperparameters which lead to finding the local minimum. It prints out Figures from 4 to 7 and the 2D representation of the cost surface with varying beta and rho parameters.'

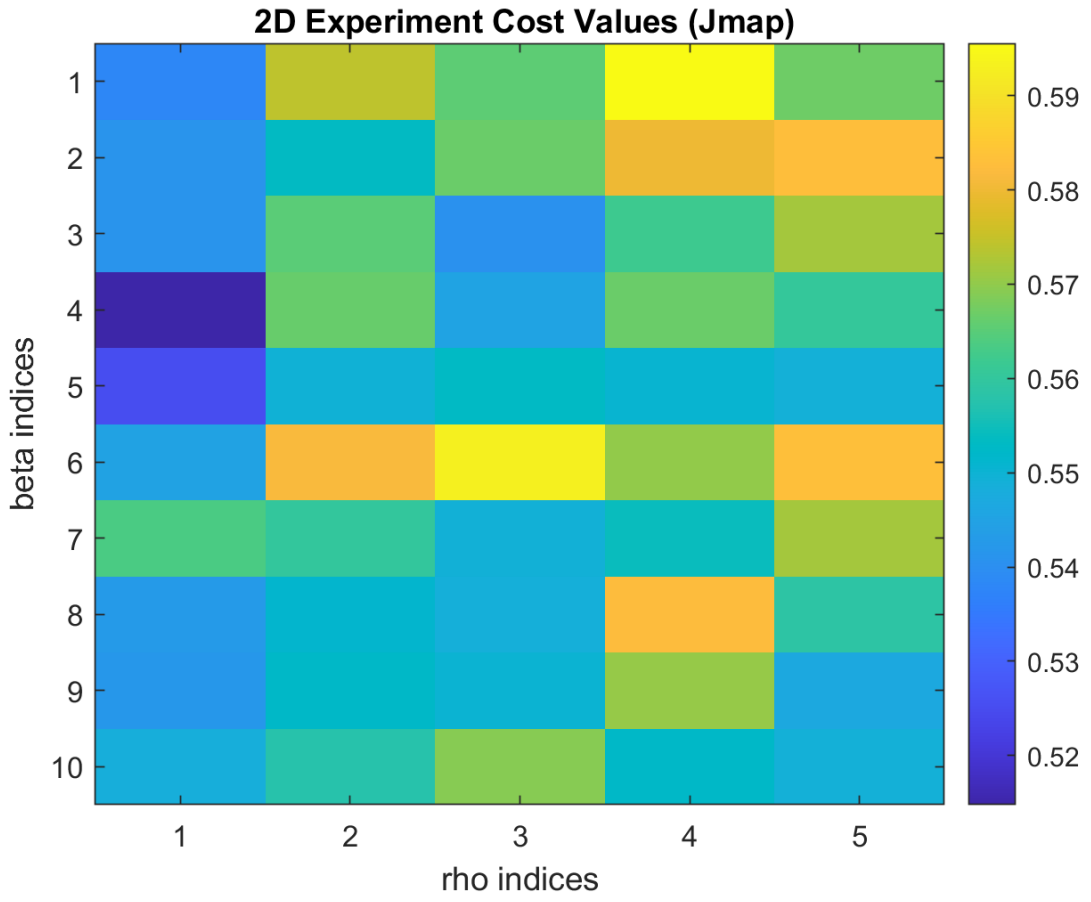


Figure 8. 2D Cost map with respect to beta and rho indices

Note that the location in dark blue is the point where the cost is minimized. The min/max range is between 0.52 and 0.60 around these values. For reference the cost for each experiment is listed in the command window. I did not put the results here as there are many lines present.

Part C:

From my experiments I have seen that the best quadruple parameters become the following:

- $L_{in} = 256$
- $L_{hid} = 64$
- $\lambda = 5 \text{ X } 10^{-4}$
- $\beta = 0.001$
- $\rho = 0.25$

Using these parameters, I have written this code and it outputs the figure below:

```
%% Part C:
paramStar = struct('Lin', 256, 'Lhid', 64, 'lambda',5e-4, ...
    'beta', betas(betastar), 'rho', rhos(rhostar));

% Deconstruct best weights - already calculated:
[w1st, w2st, ~, ~] = unwrapper(weStar, paramStar.Lin, paramStar.Lhid);

% Display weights:
dispWeights(w1st, 8,8);
```

Here 'dispWeights' is a method which reshapes unwrapped weight matrix and plots them in a subplot with appropriate title.

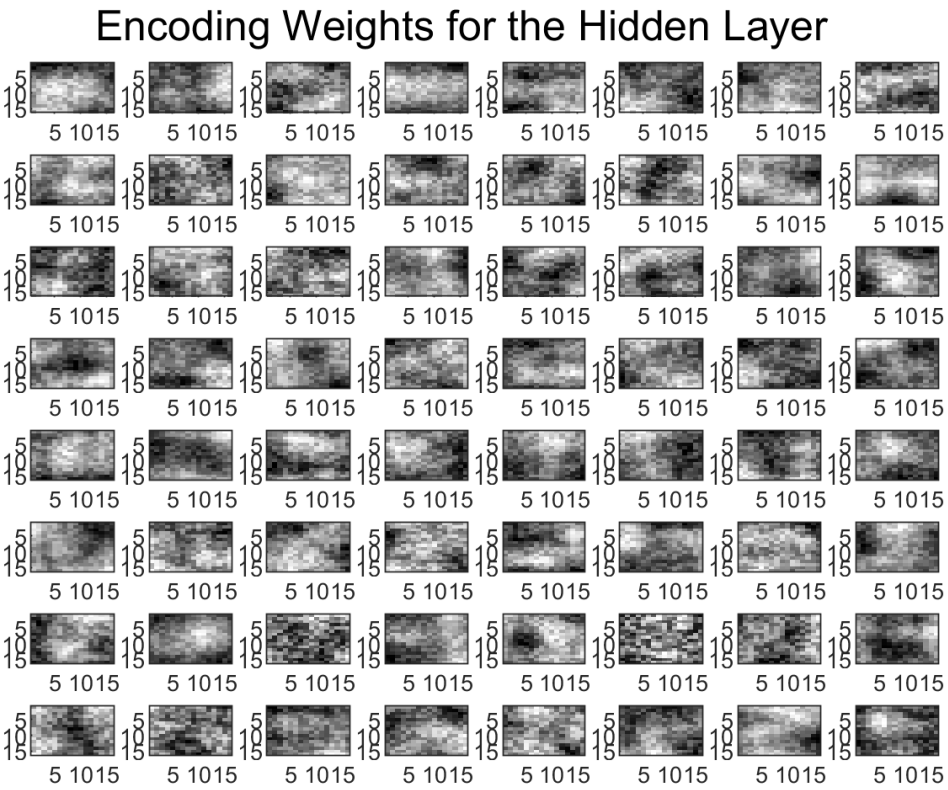


Figure 9. Encoding Weights for the hidden layer features

Since there were 64 hidden units in Part B, there are 64 images in Figure 9. These image patches are somewhat like original gray-scale images with some loss of contrast and addition of noise. These patches indicate that the hidden layer was able to compress and learn important features that reside in the grayscale dataset. The images look like face patches. For example in row 7 column 2, I see a forehead and eyebrow.

Part D:

This part is very similar to Part B. This time experiments are done on number of hidden units present in the autoencoder as well as the lambda parameter, where beta and rho are set to ideal pair of parameters found in Part B.

The code is given below but I will not go into details as implementation is very similar to Part B.

```
%% Part D:
% Retrain for different values: - 9 experiments in total
Lhid_exp = linspace(10,100,3);
lambda_exp = linspace(0,0.10e-3,3);
[params_d] = generateParamSet(Lhid_exp, lambda_exp, ...
    bestBeta, bestRho , []);

% Outputs cell array to deal with multi-dim array of different sizes.
[weightsAll] = partD_experiments(params_d, ndata);

% Draw images:
for i = 1:size(params_d,2)
    dispWeights(weightsAll{i},ceil(sqrt(params_d(i).Lhid)), ...
        ceil(sqrt(params_d(i).Lhid)));

    % Beautify plot by adding title: - overrides old title.
    sgtitle(['Hidden layer features with lambda = ', ...
        num2str(params_d(i).lambda), ...
        ' and Lhid = ', num2str(params_d(i).Lhid)]);

end
```

This code both prints out the cost associated with each experiment and shows the image patches respectively.

```
Doing a grid search over Hidden layer size and lambda values:
Doing experiments, please wait ... 1/9.
Cost value = 0.5812.
Doing experiments, please wait ... 2/9.
Cost value = 0.59217.
Doing experiments, please wait ... 3/9.
Cost value = 0.55361.
Doing experiments, please wait ... 4/9.
Cost value = 0.4409.
Doing experiments, please wait ... 5/9.
Cost value = 0.44999.
Doing experiments, please wait ... 6/9.
Cost value = 0.47082.
Doing experiments, please wait ... 7/9.
Cost value = 0.37286.
Doing experiments, please wait ... 8/9.
Cost value = 0.37072.
Doing experiments, please wait ... 9/9.
Cost value = 0.39208.
```

Figure 10. Experiment Costs

In Figure 10, there are 9 experiments since there are 3 different number of hidden layers and three number of lambdas available. The layer size – lambda value pair are given in the experiment order shown above.

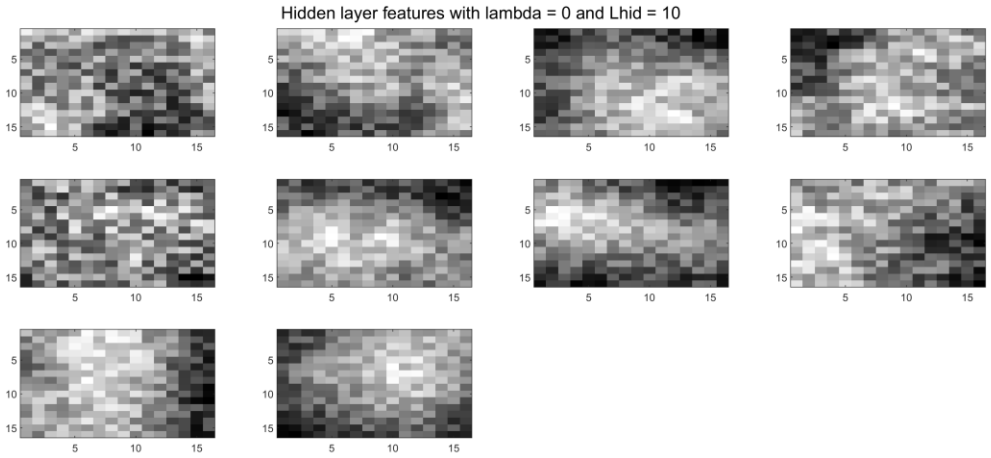


Figure 11. Experiment 1

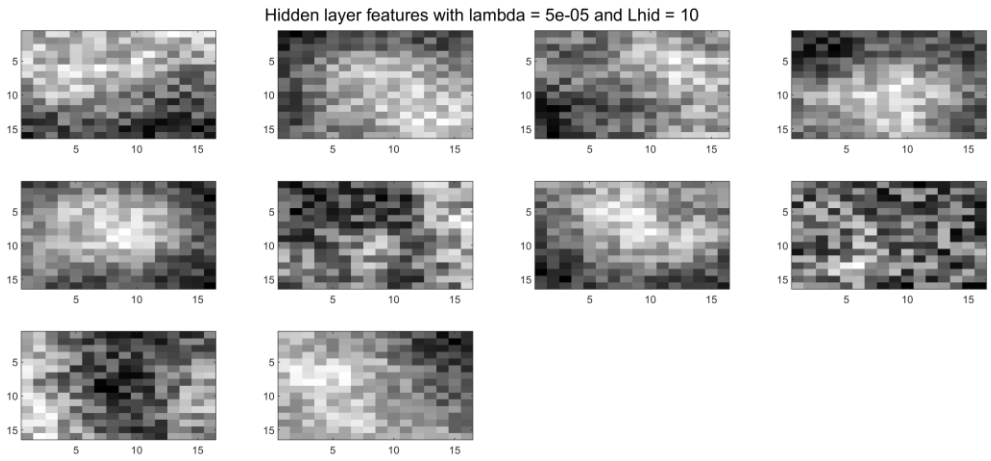


Figure 12. Experiment 2

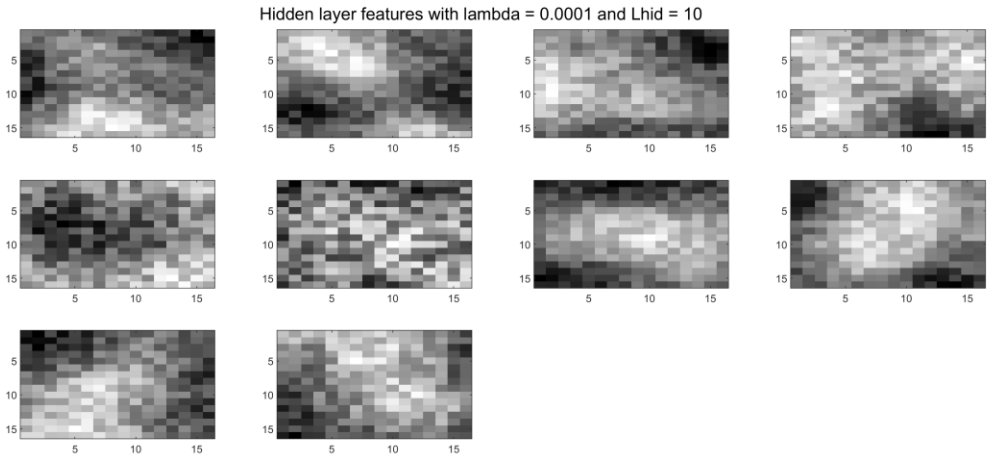


Figure 13. Experiment 3

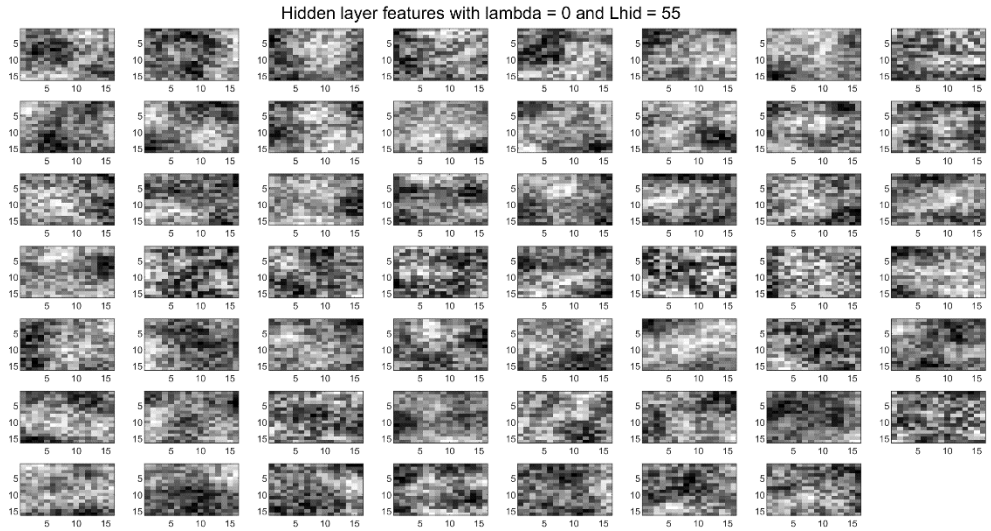


Figure 14. Experiment 4

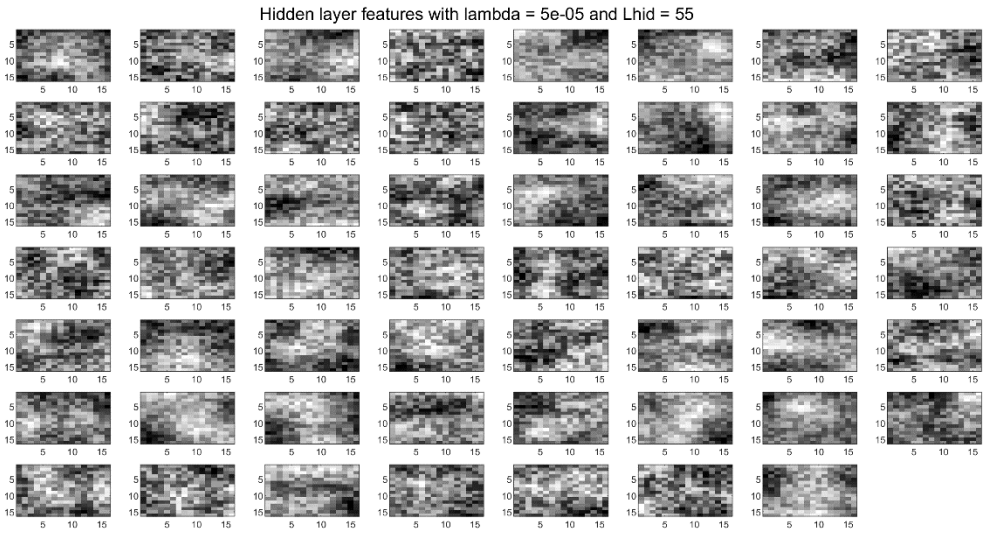


Figure 15. Experiment 5

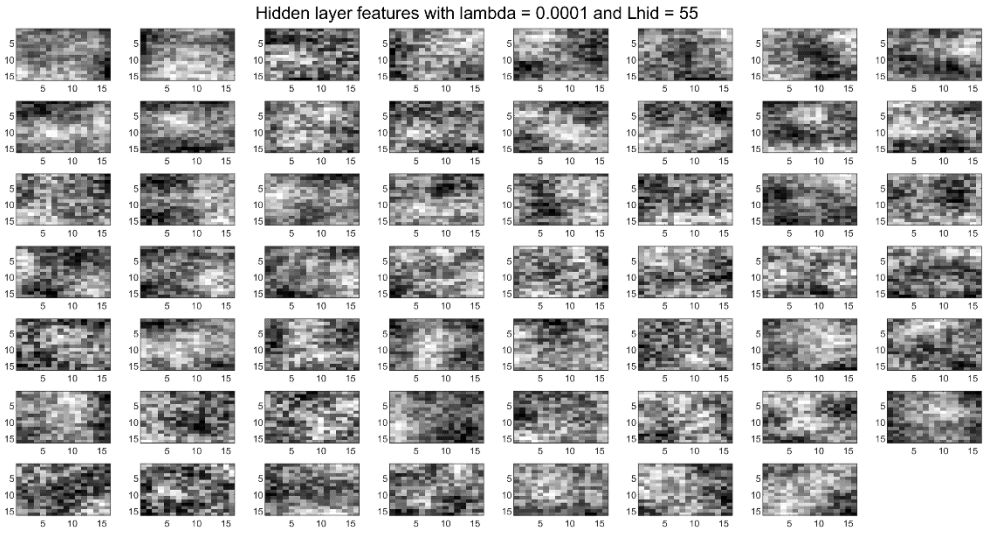


Figure 16. Experiment 6

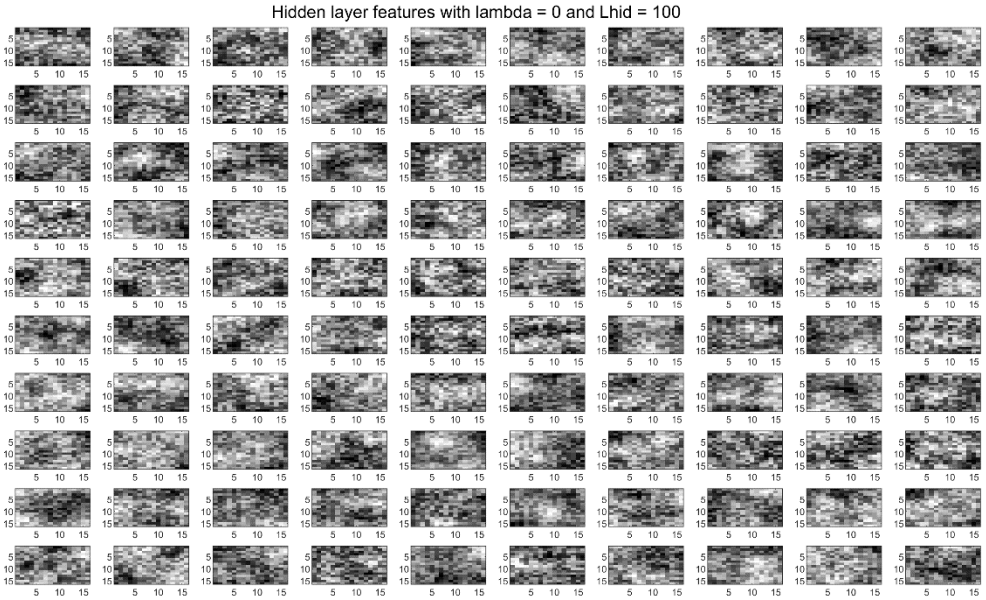


Figure 17. Experiment 7

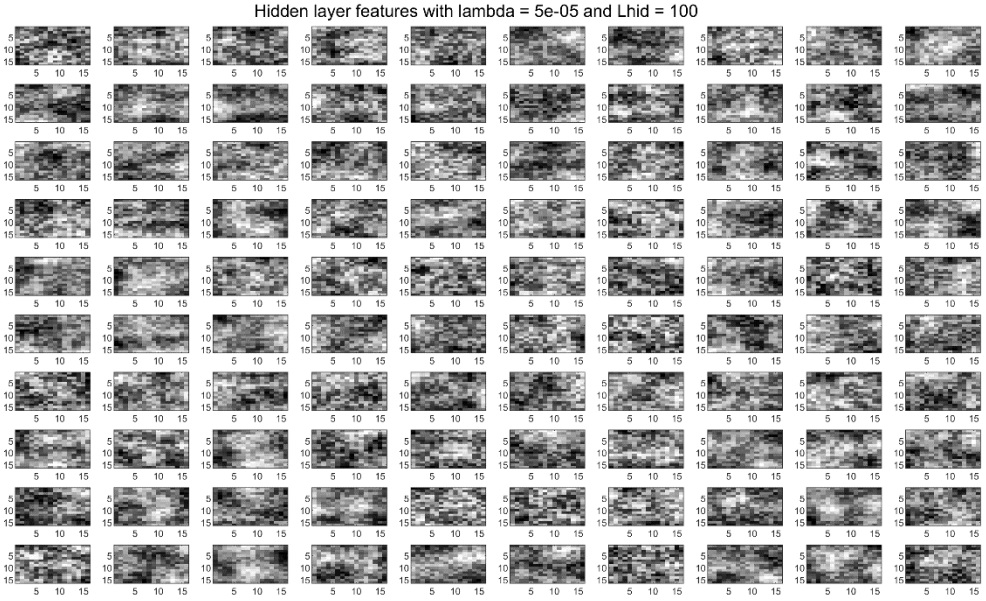


Figure 18. Experiment 8

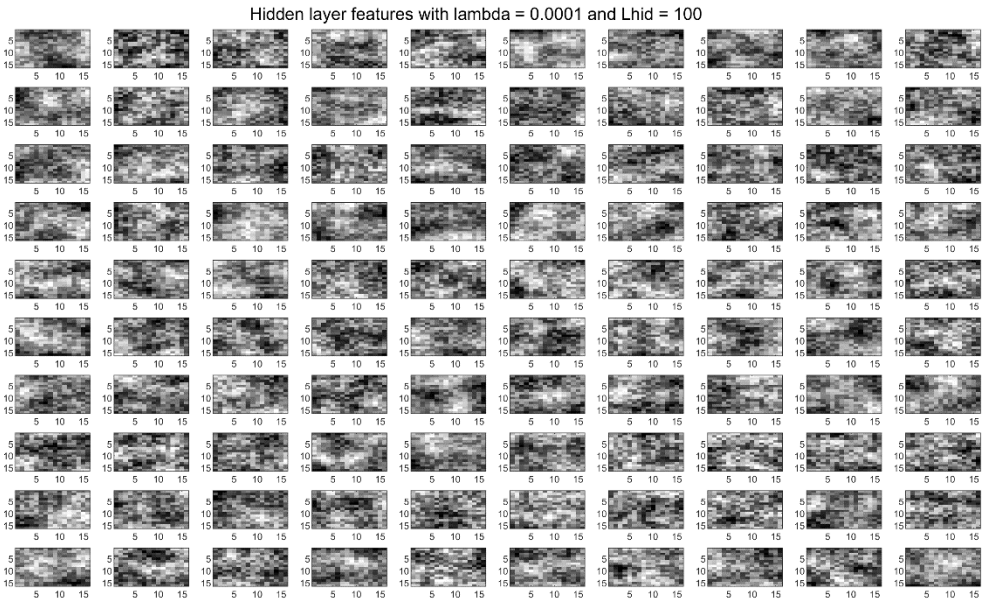


Figure 19. Experiment 9

Using Figure 10, I know that the best performance is achieved by experiment 8. It indicates that large number of hidden layers reduce the cost most efficiently. It is reasonable given that if the size of the hidden layer is the same as the input layer, hidden layer does not have to compress the information, retaining the most important features. Given no compression, there is no loss and therefore it is easier for cost to decrease.

Lambda controls the regularization term. This result indicates the regularization should be selected in the middle ground. It would be better if, more refined experiments were made on the model parameters.

Looking at the images, significantly reducing the hidden layer size affect the hidden layer weights badly as the hidden layer features were able grasp very little number of natural images. As we increase on medium range hidden layer size, in experiment 5, parts of human faces are apparent in the images. The images are noisy however the patches are more easily distinguishable. As the number of hidden layers are kept increasing, a greater number of different natural image patches (face patches) are recognized however some of the images become indistinguishable, implying those neurons weren't able to extract useful information from the dataset.

These experiments indicate that for this single hidden layer auto encoder, the number of hidden layers must be kept at a decent middle ground value, so does the lambda. Too much number of hidden units cause sparsity and noise, whereas too little hidden units cannot be trained efficiently and information in the data is lost.

Question 2:

Note: In this question, I will be using the demo codes provided with the assignment. This part does not include MATLAB code and is separate from the implementation of Question 1. Some code is altered or re-written to meet with Question 2's requirements. These changes can be seen in the appendix and it is also commented in the explanation below.

Part A:

In this part I will work with the Python notebook "Convolutional_Networks.ipynb". Since the code is already written I will go over what is implemented in the code from the PDF file that I have generated after running the code. I will take screenshots from this PDF file and explain as I go along.

Note that there were some deprecated methods which forced me to change some of the source code. I will also talk about them briefly. The entire Python notebook can be found at the end of this report.

Before I started running the cells, I had to execute the following code, using a terminal inside Jupyter application to compile some codes using Cython. These codes were necessary to use "fast layers" which provide faster computations for the convolutional neural networks:

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

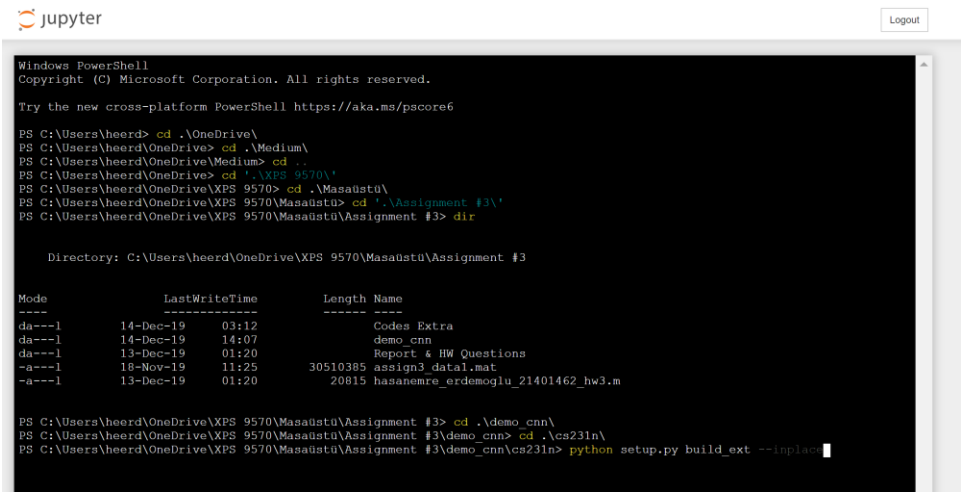


Figure 20.Implementation on Jupyter Terminal

I also had to chance the setup.py which calls Cython to compile necessary files as Python 3.7 was not compatible with Visual Studio's compiler. A try-catch phrase did the trick:

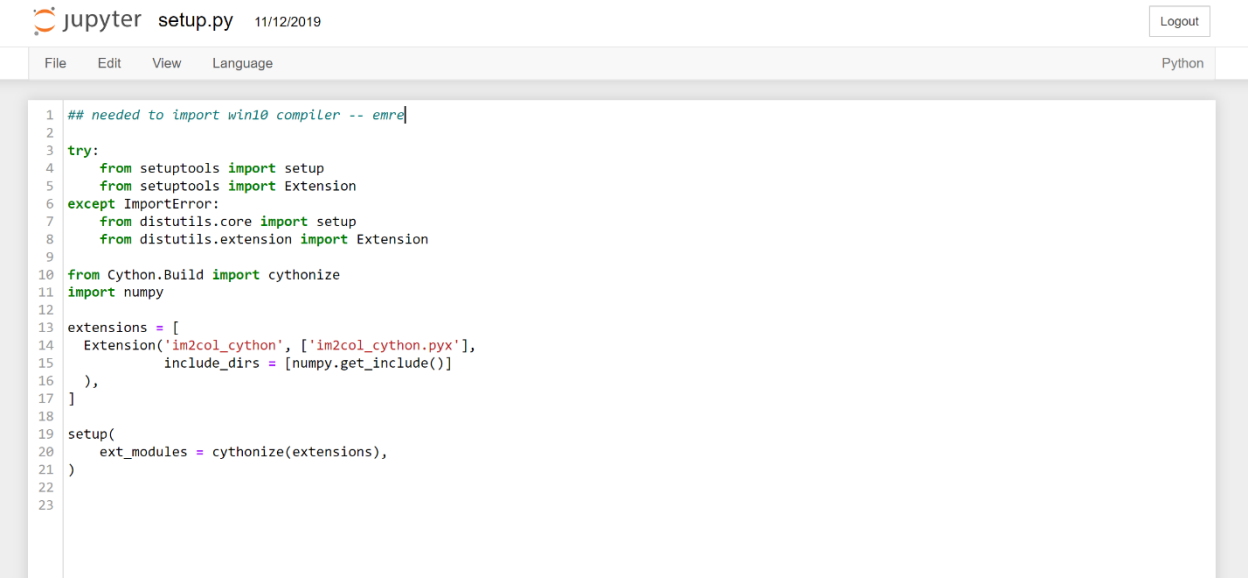


Figure 21. Try-Except is used to force Win10 compiler to work through different libraries

In this part of the homework, I am using Anaconda with Python 3.7 installed. Scipy's 'imread' was deprecated at this point and it was no longer useable. This method is moved to imageio library, so I had to change data_utils.py file line 7 to fix issues in the notebook.

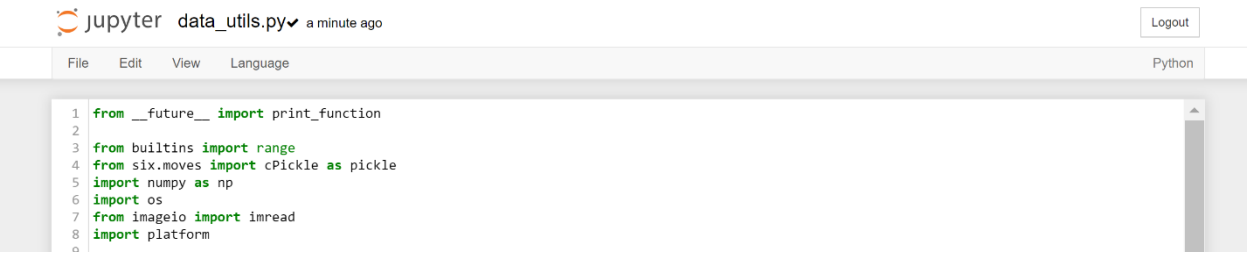


Figure 22. Data utils file is responsible for loading and displaying images from CIFAR 10 set

In the notebook, for all imreads; I have used imageio library instead of scipy. For all transform methods, I have called skimage instead of scipy; due to the scipy methods no longer being supported.

```
#from scipy.misc import imread, imresize # imread and imresize deprecated use
 below -- emre
from imageio import imread
import skimage

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, :, :, :] = skimage.transform.resize(puppy, (img_size, img_size)).transpos
e((2, 0, 1))
x[1, :, :, :] = skimage.transform.resize(kitten_cropped, (img_size, img_size))
.transpose((2, 0, 1))

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

Figure 23. Use of skimage

In the first part of the assignment, the assignment talks about inefficient and naïve implementations of forward and backward propagation in convolutional layers and max-pooling layers. In code cells [1] and [2] (from now on I will tell them as code cells 1 and 2, without using the brackets) setup and data loading was made.

In code cell 3, the program calls conv_forward_naive to do the convolution operation. This code is found within layers.py within cs231n file and can be found below:

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

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

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

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

    Returns a tuple of:
    - out: Output data, of shape (N, F, H', W') where H' and W' are given by
      H' = 1 + (H + 2 * pad - HH) / stride
      W' = 1 + (W + 2 * pad - WW) / stride
    - cache: (x, w, b, conv_param)
    """
    out = None
    ###########################################################################
    # TODO: Implement the convolutional forward pass.                         #
    # Hint: you can use the function np.pad for padding.                      #
    ###########################################################################

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

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

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

    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = (x, w, b, conv_param)
    return out, cache
```

Figure 24. Naive forward convolution operation

This code does the 2D convolution and is already implemented and was put here for reference. I have done this for every method which is on this notebook. This code was used in images of a cat and dog. First the colored images are filtered and processed to a grayscale image and then passed to the convolution layer. At the end convolutional layer outputs, the edges of the image. The outputs are shown below:

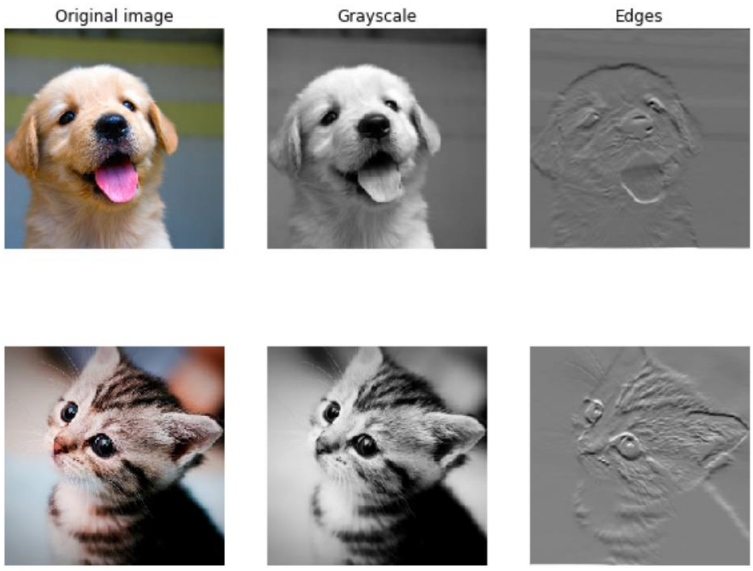


Figure 25. Naive Convolution Toy Example Outputs

Similarly, a naïve backward implementation was also implemented within layers.py. It is also used to check the gradients. Outputs are available under cell 4 in the appended PDF.

In convolution, when filter is passing by, generating the hidden units, the neighboring pixels which are multiplied and summed with the filter, tend to tie individual hidden layer results together. Convolution operation is a moving window weighting operation with a given patch size. Due to this reason, in backpropagation, the derivatives are also tied together at neighboring filter results. In backpropagation implementation this effect was observed in the code.

Max-pooling is used to down sample the number of hidden layers generated by the image. It is applied to non-overlapping regions of patches (defined by the filter size). It is used to reduce overfitting. Along with the convolution operation, max pooling provides the neural network with scale and location invariance in the image. This means that if the network has to detect a dog face, given the face is visible in the image, the network will be able to detect the face given any size and location the face is located at.

For the backpropagation of the max-pooling layer, only the hidden unit selected as the winning unit is multiplied with the derivative where there are no chances made on the other hidden units.

In the second part of the notebook, the author talks about fast_layers.py, which are the efficient implementations of the previous naïve methods. The operations are the same, but the time complexity of the operations is vastly reduced. This part uses the libraries compiled by Cython and the details of the implementations are hidden. However, it is noted that the fast implementation is fast it exploits non-overlapping pooling regions and given the images are of the same size. If these conditions are not met, the performance is very close to the naïve methods.

The dataset we are given is indeed meeting up with the expectations, as there is a 275 times increased performance with almost no difference in the computational results.

Before starting on a structured example, the notebook talks about sandwich layers which are nothing but pre-defined set of layers. Inside layers_utils.py, some examples are given.

```
def conv_relu_forward(x, w, b, conv_param):
    """
    A convenience layer that performs a convolution followed by a ReLU.

    Inputs:
    - x: Input to the convolutional layer
    - w, b, conv_param: Weights and parameters for the convolutional layer

    Returns a tuple of:
    - out: Output from the ReLU
    - cache: Object to give to the backward pass
    """
    a, conv_cache = conv_forward_fast(x, w, b, conv_param)
    out, relu_cache = relu_forward(a)
    cache = (conv_cache, relu_cache)
    return out, cache


def conv_relu_backward(dout, cache):
    """
    Backward pass for the conv-relu convenience layer.
    """
    conv_cache, relu_cache = cache
    da = relu_backward(dout, relu_cache)
    dx, dw, db = conv_backward_fast(da, conv_cache)
    return dx, dw, db
```

Figure 26. Sandwich Layer Example

The code above gives an example for such implementation. For example, conv_relu_forward first implements a “fast” convolution implementation of the data or previous hidden units, then it feeds the results to a ReLu and then returns the results. These can be stacked in a model to develop any model that we like. This type of implementation groups layers together so when we are building the model the entire structure is more organized and visible.

Finally, the notebook talks about three-layer Convolutional Networks. The model is described by a file named cnn.py with class name ThreeLayerConvNet. In this code, the loss for the 3-layer convolutional network and its gradient as well as the method which initializes the network is defined. The bulk of the code can be seen below:

```
def loss(self, X, y=None):
    """
    Evaluate loss and gradient for the three-layer convolutional network.

    Input / output: Same API as TwoLayerNet in fc_net.py.
    """
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    W3, b3 = self.params['W3'], self.params['b3']

    # pass conv_param to the forward pass for the convolutional layer
    # Padding and stride chosen to preserve the input spatial size
    filter_size = W1.shape[2]
    conv_param = {'stride': 1, 'pad': (filter_size - 1) // 2}

    # pass pool_param to the forward pass for the max-pooling layer
    pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

    scores = None
    ############################################################################
    # TODO: Implement the forward pass for the three-layer convolutional net,  #
    # computing the class scores for X and storing them in the scores          #
    # variable.                                                                #
    #                                                                          #
    # Remember you can use the functions defined in cs231n/fast_layers.py and  #
    # cs231n/layer_utils.py in your implementation (already imported).         #
    ############################################################################

    conv_relu_pool_out, conv_relu_pool_cache = conv_relu_pool_forward(X, W1, b1,
        conv_param, pool_param)
    affine_relu_out, affine_relu_cache = affine_relu_forward(conv_relu_pool_out,
        W2, b2)
    scores, scores_cache = affine_forward(affine_relu_out, W3, b3)

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

    if y is None:
        return scores

    loss, grads = 0, {}
    ############################################################################
    # TODO: Implement the backward pass for the three-layer convolutional net, #
    # storing the loss and gradients in the loss and grads variables. Compute  #
    # data loss using softmax, and make sure that grads[k] holds the gradients #
    # for self.params[k]. Don't forget to add L2 regularization!               #
    #                                                                          #
    # NOTE: To ensure that your implementation matches ours and you pass the   #
    # automated tests, make sure that your L2 regularization includes a factor #
    # of 0.5 to simplify the expression for the gradient.                      #
    ############################################################################

    loss, dscores = softmax_loss(scores, y)
    loss += 0.5 * self.reg * (np.sum(W1*W1) + np.sum(W2*W2) + np.sum(W3*W3))

    daffine_relu_out, dW3, db3 = affine_backward(dscores, scores_cache)
    dconv_relu_pool_out, dW2, db2 = affine_relu_backward(daffine_relu_out, affine_relu_cache)
    dx, dW1, db1 = conv_relu_pool_backward(dconv_relu_pool_out, conv_relu_pool_cache)

    grads['W1'] = dW1 + self.reg * W1
    grads['W2'] = dW2 + self.reg * W2
    grads['W3'] = dW3 + self.reg * W3
    grads['b1'] = db1
    grads['b2'] = db2
    grads['b3'] = db3

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

    return loss, grads
```

Figure 27. Three-layer Convolutional Network Loss

This loss function realizes the network using 1 convolutional ReLu network with pooling layer, then follows by an affine ReLu layer finalized by a final ReLu layer. Affine_relu_forward first calculates the outputs for a fully connected layer, followed by a ReLu layer.

Conv_relu_pool_forward, first calculates the fast-convolutional layer, then it implements a relu operation on the outputs layer implements max-pool on the outputs of the relu function. Vanilla affine_forward simply calculates the outputs given the weights and the input as a fully connected layer.

Looking at the big picture we have the following chain: Fast-Conv -> ReLu -> Fast-Pool -> Fully Connected -> ReLu -> Fully Connected. It looks like there are more than thre layers, but when the operations are combined there are Convolutional layer followed by Fully Connected ReLu and Fully Connected Layers.

The code does sanity check on loss values to use display whether something is wrong with the initialization, then it does numeric gradient check to see whether the derivative of the loss function is calculated correctly.

This model is trained with little data to train convolutional kernels which overfit to the data. As expected overfitting occurs:

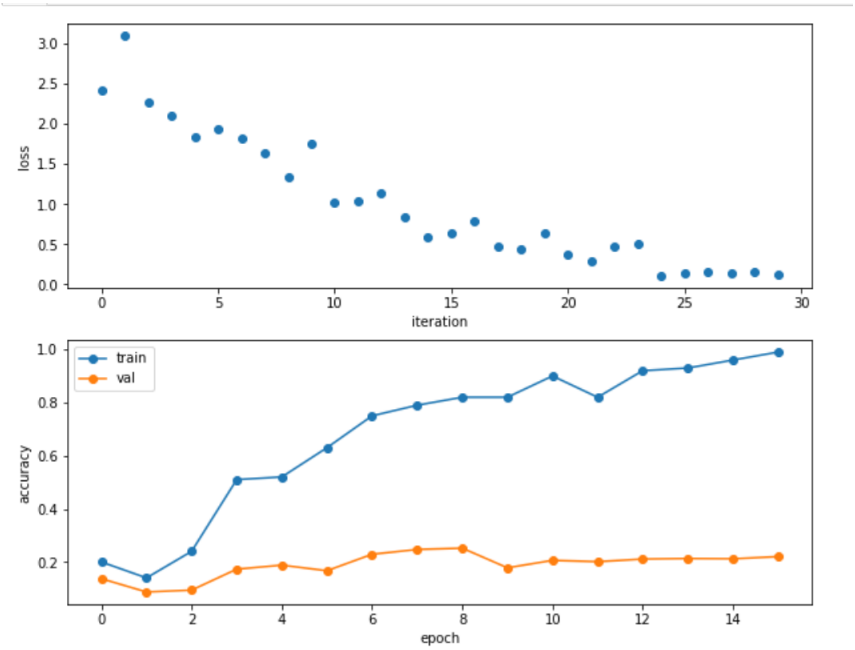


Figure 28. Training over Epoch on small data size

Overfitting is visible since the loss decreases over time where training accuracy keeps increasing without any increase in test accuracy. The model is able to predict correct labels to the objects which it has seen before, but it fails at classifying images which are previously not available. With large amount of data, the test accuracy is able to go up to 40 percent with a single epoch as the model had more examples to train on. This increase in test accuracy is due to variety of boundary cases that the model has seen in the training stage. More boundary cases that a model encounter better it responds to cases which are closer to these boundary cases.

The code also visualizes the convolutional filters after training to express, which filters did the model learn in order to achieve this classification results. Fully connected layers after this convolutional layer use these kernels on the hidden units to identify features in the provided images to do classification on the data.

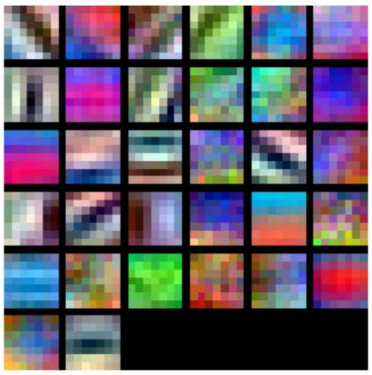


Figure 29. Trained Convolution Kernels

Batch normalization is a vital technique for deep fully connected networks to overcome effects of vanishing gradients and exploding gradients by containing the input and hidden unit outputs within a certain range which does not cause very small or very large gradients. In Convolutional Neural Networks, this operation has to be done spatially. The notebook finally covers this topic.

The data is taken as tensors (N,C,H,W). N stands for the batch size, C stands for channel size (3 for RGB channel). The spatial dimensions are denoted by H,W which are height and width respectively. The spatial batch normalization is done across each channel. Mean and variance is calculated individually for each pixel described by H and W along all samples of the minibatch.

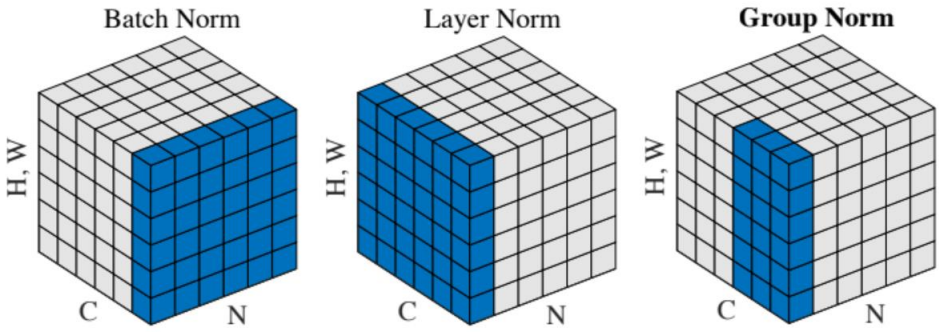


Figure 30. Comparison of Normalization Techniques

The image above is very descriptive about how normalization is applied over the tensors. In batch normalization, the tensor on the left, we normalized the data using all samples within the batch, for each channel individually where we calculated mean and variance for each pixel in H and W. In layer normalization the same operation is done with the exception that now, instead of normalizing along the items in the batch, we normalize every item individually along number of channels it has. Group normalization is doing layer normalization by dividing the channels to groups.

Part B:

For this part I have selected to work with TensorFlow. Due to this reason I will work with the Python notebook called "TensorFlow.ipynb". Since TensorFlow was downloaded with Anaconda3, I did not need to download and install additional files. Again, the entire implementation added to the appendix, I will go over the tutorial that the notebook provides and the parts that I have written in order to complete the tutorial. This tutorial is well designed in the way that it provides you resources to research about the topic and example implementation which you can mimic to answer tutorial implementations.

This code has 5 parts, where first part is the preparation of CIFAR 10 dataset which is the dataset used for this training session. Other 4 parts start from low-level API implementations to high-level implementations sequentially. In the final part I will try to implement my own network using TensorFlow to experiment on what I have learned.

Note that for this part, the explanations have the codes in screenshots, however the actual code in text form is available in the appendix.

I have used version 1.14.0 of TensorFlow for compatibility, as version 2.0 has some of the methods deprecated. I will skip the preparation part as it just introduces size of the train, validation and test data and labels and a Dataset object, which is an iterable object. I also set my device's GPU to be True, therefore I will be using my GPU while training my networks.

Barebone TensorFlow offer most flexibility in defining and compiling neural networks at the cost least convenience in terms of readability and complexity of the code. It is a low-level API. TensorFlow deals with static computational graphs, in our case these graphs are the networks that we want to implement. The operations on these computational graphs are done using tensors, which TensorFlow is named after.

TensorFlow handles these operations in two steps, first a computational graph is defined, in this stage no calculations are made. In the second stage, this graph is run under given constraints, optimizers, loss functions, etc. One should note that default tensor placement is ordered as batch_index, height_index, width_index, channel_index in TensorFlow, unless explicitly stated. Other libraries may have different representations of the tensors.

In flatten part of the tutorial code, it is seen that TensorFlow flattens out the data in last three dimensions, that is the height, width and channel. The flattened output is given per each data sample. Since flatten works this way, it is vital to have the batch index as the first index of the tensor.

```
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.]]
```

Figure 31. Flatten Examples

The tutorial gives example for 2 fully connected case for Barebone TensorFlow tutorial. First a two layer computational graph is implemented, where x is the tensor data and params is the weights which are packed into a single matrix. First the tensor is flattened, then it is multiplied with 1st layer weights, passed through ReLU and passed through another set of weights as the second layer. This method directly gives the forward computational graph and defined by the name "two_layer_fc". Its test methods "two_layer_fc_test" constructs the computational graphs with zero weights and placeholder data. Then a TensorFlow session is opened and the data is fed to the placeholder. The output has the correct shape showing that we are on the right track. (Outputs are not important as we are not training anything with zero weights just yet.)

Barebones TensorFlow tutorial requires me to develop a 3-layer convolutional network. Using the example and the links provided, it was easy for me to generate a computational graph. The requirements were also explicitly stated.

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)

Figure 32. Barebone TensorFlow Tutorial Requirements

The following is the code I have written to define the computational graph.

```
In [12]: def three_layer_convnet(x, params):
    """
    A three-layer convolutional network with the architecture described above.

    Inputs:
    - x: A TensorFlow Tensor of shape (N, H, W, 3) giving a minibatch of image
s
    - params: A list of TensorFlow Tensors giving the weights and biases for t
he
      network; should contain the following:
      - conv_w1: TensorFlow Tensor of shape (KH1, KW1, 3, channel_1) giving
        weights for the first convolutional layer.
      - conv_b1: TensorFlow Tensor of shape (channel_1,) giving biases for the
        first convolutional layer.
      - conv_w2: TensorFlow Tensor of shape (KH2, KW2, channel_1, channel_2)
        giving weights for the second convolutional layer
      - conv_b2: TensorFlow Tensor of shape (channel_2,) giving biases for the
        second convolutional layer.
      - fc_w: TensorFlow Tensor giving weights for the fully-connected layer.
        Can you figure out what the shape should be?
      - fc_b: TensorFlow Tensor giving biases for the fully-connected layer.
        Can you figure out what the shape should be?
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None
    ############################################################################
##
    # TODO: Implement the forward pass for the three-layer ConvNet.
#
    ############################################################################
##
    layer1_conv = tf.nn.conv2d(input = x,filter = conv_w1,
                               strides = [1,1,1,1], padding = 'SAME', name =
'conv1') + conv_b1
    layer1_conv_relu = tf.nn.relu(layer1_conv)

    layer2_conv = tf.nn.conv2d(input = layer1_conv_relu,filter = conv_w2,
                               strides = [1,1,1,1],padding = 'SAME' ,name = 'c
onv2') + conv_b2
    layer2_conv_relu = tf.nn.relu(layer2_conv)

    layer3 = flatten(layer2_conv_relu)

    scores = tf.matmul(layer3,fc_w) + fc_b
    ############################################################################
##
    #                               END OF YOUR CODE
#
    ############################################################################
##
    return scores
```

Figure 33. Three Layer Conv-Net

The test function for this method is already written written and outputs (64,10) for the shape of the scores which is correct stated by the notebook.

Later the notebook proceeds to the training step using TensorFlow. In this part there is no parts that I need to code by myself. However, the code is provided for me to follow it.

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

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

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

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

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

Figure 34. Training Step for Barebone Tensorflow Tutorial

In this code first a cross entropy loss is defined, then this loss is averaged across the minibatch. Derivative computations are added by tf.gradients. This function traverses the provided computational graph internally to come up with the symbolic derivatives so that the user only must

provide the forward propagation. Backpropagation is handled by the TensorFlow. Finally, it requires a control dependency to update the network weights each time a loss is evaluated.

In the train_part2 method, a model on CIFAR-10 dataset is evaluated. Computational graph has been cleared so repeated runs of the same cell would cause duplicate additions of the same computational graph many times. Then, placeholders are described for data and labels and necessary model and loss functions are described using the placeholder. Finally, a TensorFlow session is run while feeding the CIFAR-10 data and labels in their respective placeholders to train the model. I will not add a screenshot of this section as it is already written inside the notebook.

Note that up until this stage, we wrote function which would compile the loss, gradients, training loop and other necessary components that TensorFlow requires to compute the network. After these steps are done, the notebook writes a Kaiming normal initializer for the weights, it generates the set of weights for 2 layer case, then runs the

Note that up until this stage, we wrote function which would compile the loss, gradients, training loop and other necessary components that TensorFlow requires to compute the network. After these steps are done, the notebook writes a Kaiming normal initializer for the weights, it generates the set of weights for 2 layer case, then runs the model using train_part2 methods which constructs the network and the TensorFlow session. The results for this 2-layer network is given below:

```
In [18]: def two_layer_fc_init():
             """
             Initialize the weights of a two-layer network, for use with the
             two_layer_network function defined above.

             Inputs: None

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

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

Iteration 0, loss = 3.1610
Got 108 / 1000 correct (10.80%)
Iteration 100, loss = 1.8886
Got 375 / 1000 correct (37.50%)
Iteration 200, loss = 1.5118
Got 371 / 1000 correct (37.10%)
Iteration 300, loss = 1.7202
Got 378 / 1000 correct (37.80%)
Iteration 400, loss = 1.7810
Got 408 / 1000 correct (40.80%)
Iteration 500, loss = 1.7605
Got 434 / 1000 correct (43.40%)
Iteration 600, loss = 1.9183
Got 421 / 1000 correct (42.10%)
Iteration 700, loss = 1.9711
Got 448 / 1000 correct (44.80%)
```

Figure 35. Two-layer case Initialization & Running

The tutorial requires me to construct the same system for three-layer convolutional network with these specifications:

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.

Using the guidance given within the code, I have written the following code:

In [19]:
```
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)
```

Figure 36. Three Layer Conv-Net training with Barebone TensorFlow

Since train_part2 is generic, the code is used to initiate the training the following is the results. Note that the accuracies a bit lower than what the notebook suggests, the implementation of the code is correct and at each run of this code, accuracies vary a little. In my own experiment I was able to reach up to around 40 percent accuracy for this question.

```
Iteration 0, loss = 3.0160
Got 95 / 1000 correct (9.50%)
Iteration 100, loss = 1.9559
Got 300 / 1000 correct (30.00%)
Iteration 200, loss = 1.8292
Got 309 / 1000 correct (30.90%)
Iteration 300, loss = 2.0164
Got 296 / 1000 correct (29.60%)
Iteration 400, loss = 1.9395
Got 347 / 1000 correct (34.70%)
Iteration 500, loss = 1.9531
Got 341 / 1000 correct (34.10%)
Iteration 600, loss = 1.9506
Got 366 / 1000 correct (36.60%)
Iteration 700, loss = 1.6701
Got 355 / 1000 correct (35.50%)
```

Figure 37. Training Results for 3-layer conv-net

After completing the basics of barebone TensorFlow, the notebook explains the Keras Model API, which is less complex, as we are not required to manually keep track of all tensors and parameters and the code is easier to read.

After reading the basics and some vital implementations of the Model API the notebook proceeds with two -layer network as a tutorial example and wants user to implement the same network in Part 2 – Barebone TensorFlow, but this time using Model API.

The key parts of the Model API are the following:

- One should subclass Keras Model API in order to use the model API. The hidden variables of the Model API is required in my network so they must be initialized using super().__init() method.
- Tf.variance_scaling_initializer is an initializer that is required to initialize trainable parameters for the network. Details were given with a link to TensorFlow's webpage.
- Call() methods, describes the computational graph that one needs to describe and implements the forward pass. Tensorflow already handles with the backward pass given forward pass.

I will pass two-layer-fc example network (Both Module and Functional API) and proceed with the network I have implemented as the implementation roots from these example steps. Functional API takes every layer as a function, whereas Module API takes entire network as an object. That is, in module API, we define a class for the network, whereas in functional API the network is described as a method.

The following is the requirements for Three-Layer ConvNet with support material.

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)

I have used the information above to implement the following:

In [22]:
```
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.
#
        ########################################################################
##
        # 5x5 kernels, relu non-linearity
        var_init = tf.variance_scaling_initializer(scale=2.0)
        self.conv1 = tf.layers.Conv2D(filters = channel_1,kernel_size = [5,5],
                                      strides = [1,1],padding = 'SAME',activat
ion = tf.nn.relu,
                                     use_bias = True,kernel_initializer = var_
init,
                                      bias_initializer = var_init,name = 'conv
1')

        # 3x3 kernels, relu non-linearity
        self.conv2 = tf.layers.Conv2D(filters = channel_2,kernel_size = [3,3],
                                      strides = [1,1],padding = 'SAME',activat
ion = tf.nn.relu,
                                     use_bias = True,kernel_initializer = var_
init,
                                      bias_initializer = var_init,name = 'conv
2')
        # Fully connected layer
        self.fc = tf.layers.Dense(units = num_classes,use_bias = True,
                                  kernel_initializer = var_init,bias_initializ
er = var_init,
                                  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_c1 = self.conv1(x)
        x_c2 = self.conv2(x_c1)
        x_c2_flat = tf.layers.flatten(x_c2)
        scores = self.fc(x_c2_flat)
        ############################################################################
```



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

Module API is used here as the template is given in this form. Test results suggest the size of scores matrix is of 64 by 10, which is in line with the previous implementations. This means the implementation is correct.

The training function for the Model API, train_part34 is used for both Model and Sequential API's of TensorFlow. It instantiates the placeholders for the models, defines cross entropy loss, defines an optimizer and control dependencies to setup gradient update mechanisms. Later it opens a TensorFlow session to run the model, provided necessary parameters.

For both Module & Functional API the notebook does training and provides the output

```
Starting epoch 0
Iteration 0, loss = 2.8361
Got 141 / 1000 correct (14.10%)

Iteration 100, loss = 1.8689
Got 393 / 1000 correct (39.30%)

Iteration 200, loss = 1.4486
Got 388 / 1000 correct (38.80%)

Iteration 300, loss = 1.7736
Got 377 / 1000 correct (37.70%)

Iteration 400, loss = 1.8063
Got 442 / 1000 correct (44.20%)

Iteration 500, loss = 1.9122
Got 432 / 1000 correct (43.20%)

Iteration 600, loss = 1.8056
Got 436 / 1000 correct (43.60%)

Iteration 700, loss = 1.9277
Got 444 / 1000 correct (44.40%)
```

Figure 38.Module API training

```
Starting epoch 0
Iteration 0, loss = 2.9702
Got 117 / 1000 correct (11.70%)

Iteration 100, loss = 1.9470
Got 394 / 1000 correct (39.40%)

Iteration 200, loss = 1.4413
Got 379 / 1000 correct (37.90%)

Iteration 300, loss = 1.8021
Got 368 / 1000 correct (36.80%)

Iteration 400, loss = 1.8578
Got 426 / 1000 correct (42.60%)

Iteration 500, loss = 1.7655
Got 429 / 1000 correct (42.90%)

Iteration 600, loss = 1.7955
Got 419 / 1000 correct (41.90%)

Iteration 700, loss = 1.9614
Got 444 / 1000 correct (44.40%)
```

Figure 39. Functional API training

The following is the training procedure I have written for Three-layer ConvNet:

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 [27]:
```
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,moment
um = 0.9,use_nesterov = True)
    ############################################################################
##
    #                           END OF YOUR CODE
#
    ############################################################################
##
    return optimizer

train_part34(model_init_fn, optimizer_init_fn)
```

Figure 40. Three-Layer ConvNet Training Code

The output is given below:

```
Starting epoch 0
Iteration 0, loss = 3.0034
Got 110 / 1000 correct (11.00%)

Iteration 100, loss = 1.4748
Got 453 / 1000 correct (45.30%)

Iteration 200, loss = 1.3288
Got 480 / 1000 correct (48.00%)

Iteration 300, loss = 1.4024
Got 495 / 1000 correct (49.50%)

Iteration 400, loss = 1.1305
Got 522 / 1000 correct (52.20%)

Iteration 500, loss = 1.3949
Got 554 / 1000 correct (55.40%)

Iteration 600, loss = 1.3448
Got 550 / 1000 correct (55.00%)

Iteration 700, loss = 1.2514
Got 552 / 1000 correct (55.20%)
```

Figure 41. Three-Layer ConvNet with Nesterov Momentum

The accuracies are above 45 % as told by the tutorial notebook.

The final API that this tutorial explains is the Sequential API. This API allows user to stack predefined layers one by one and due to this reason, it does not offer much flexibility. However, in terms of readability and fast deployment (no custom classes must be coded) this is the best API to use given the model intended to be implemented is very basic and consist only of stack of layers.

The training loop process is the same with the Model API so only the computational graph must be defined. Example code directly defines and runs the model using the following code:

In [28]:
```
learning_rate = 1e-2

# The tf.layers used here are deprecated so I changed them to their modern cou
nterparts.
# tf.layers.Flatten --> tf.keras.layers.Flatten

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 = 2.9567
Got 117 / 1000 correct (11.70%)

Iteration 100, loss = 1.8102
Got 386 / 1000 correct (38.60%)

Iteration 200, loss = 1.4827
Got 397 / 1000 correct (39.70%)

Iteration 300, loss = 1.8948
Got 374 / 1000 correct (37.40%)

Iteration 400, loss = 2.0211
Got 425 / 1000 correct (42.50%)

Iteration 500, loss = 1.7565
Got 434 / 1000 correct (43.40%)

Iteration 600, loss = 1.9392
Got 437 / 1000 correct (43.70%)

Iteration 700, loss = 1.9409
Got 451 / 1000 correct (45.10%)
```

Figure 42. Sequential API 2 FC Layer Example Training

The notebook also requires me to code 3-layer ConvNet using sequential API, the same network that I have previously implemented. My implementations and results are shown in the figure below. I have used the documentation on the TensorFlow page to learn about how to implement layers.

In [29]:
```
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_siz
e = [5,5],
                               strides = [1,1],padding = 'SAME',activation = tf.n
n.relu,
                               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.n
n.relu,
                               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
= initializer,
                               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 = 5e-4,momentum = 0.9
,use_nesterov = True)
    ############################################################################
##
    #                            END OF YOUR CODE
#
    ############################################################################
##
    return optimizer

train_part34(model_init_fn, optimizer_init_fn)
```

Figure 43. Sequential API, 3-Layer ConvNet Implementation

```
Starting epoch 0
Iteration 0, loss = 2.9914
Got 101 / 1000 correct (10.10%)

Iteration 100, loss = 1.7236
Got 413 / 1000 correct (41.30%)

Iteration 200, loss = 1.4269
Got 468 / 1000 correct (46.80%)

Iteration 300, loss = 1.3530
Got 480 / 1000 correct (48.00%)

Iteration 400, loss = 1.4411
Got 504 / 1000 correct (50.40%)

Iteration 500, loss = 1.5887
Got 506 / 1000 correct (50.60%)

Iteration 600, loss = 1.5389
Got 524 / 1000 correct (52.40%)

Iteration 700, loss = 1.4681
Got 542 / 1000 correct (54.20%)
```

Figure 44. Sequential Implementation Training Results

The final part of this notebook presents an Open-Ended Challenge with CIFAR-10 dataset. It requires me to implement a CNN which provides an accuracy over 70% in CIFAR-10 dataset. The notebook provides some hints and remarks about how to implement an CNN using TensorFlow and requires me to dig deeper in the API's to achieve good performance on the dataset.

I have used the original optimizer used in previous parts for optimizer_init_fn().

For model initialization, I have followed the hints that the question gave me, along with iterative design. I started with a single convolutional layer and fed it to a dense layer of size 512. Then I added a single softmax dense layer with 10 classes as we work on 10 classes in CIFAR dataset.

After implementing this network, I deepened the network by adding extra convolutional layers. I also used max pooling to add scale and position invariance, At the end of the day, I have developed the following network after numerous trials.

3 convolutions of size 3 by 3 followed by a max-pooling layer for three times, with increasing filter size 64, 128, 256, 256 respectively. All operations have ReLU activations implicitly. In total I achieved 12 convolutional layers, Then I fed the outputs to 3 dense layers of increasing size and finalized with a softmax layer of 10 neurons.

Most of the design process relied on trial and error. My idea was to extract large amount of complex features using stacks of convolutions, providing scale and position invariance by passing through max-pooling layer. At the end of these layers fully connected layers would be used to combine these convolutions and provide accurate results.

I could not use Dropout or normalization because of prewritten part of the code "train_part34", as these processes are not compatible with Keras Model or Sequential API's. Since my code worked good enough when I wanted to introduce dropout and normalization, I did not want to change the entire code therefore I didn't implement these. Using functional API then introducing dropout and batch normalization may increase the performance even more by avoiding overfitting.

The model I trained overfits around epoch, as training loss decreases but there is no significant increase in the test accuracy. Normalization and dropout possibly would have helped in my case.

In the figures below you can find the implementation I made along with training results. After 10 iterations I was above 70 % accuracy which is a good performance as it is told by the notebook. A small research suggests with more complex networks, accuracies above 90 % is possible to achieve.

In [38]:
```
def model_init_fn(inputs, is_training):
    model = None
    ############################################################################
##
    # TODO: Construct a model that performs well on CIFAR-10
#
    ############################################################################
##
    initializer = tf.variance_scaling_initializer(scale=2.0)

    model = tf.keras.models.Sequential([
    # Layer 1-3:
    tf.keras.layers.Conv2D(input_shape = (32,32,3),filters =64,kernel_size = [
3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 64,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 64,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    # Reduce dim:
    tf.keras.layers.MaxPooling2D(pool_size = [2,2]),
    #tf.keras.layers.Dropout(0.5),

    # Layer 4-6:
    tf.keras.layers.Conv2D(filters = 128,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 128,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 128,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    # Reduce dim:
    tf.keras.layers.MaxPooling2D(pool_size = [2,2]),
    #tf.keras.layers.Dropout(0.4),

    # Layer 7-9:
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
```

```
                            padding = 'SAME',activation = 'relu',
                            use_bias = True,kernel_initializer = initializer,
                            bias_initializer = initializer),
    # Layer 10-12:
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
                            padding = 'SAME',activation = 'relu',
                            use_bias = True,kernel_initializer = initializer,
                            bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
                            padding = 'SAME',activation = 'relu',
                            use_bias = True,kernel_initializer = initializer,
                            bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
                            padding = 'SAME',activation = 'relu',
                            use_bias = True,kernel_initializer = initializer,
                            bias_initializer = initializer),

    # Reduce dim:
    tf.keras.layers.MaxPooling2D(pool_size = [2,2]),
    #tf.keras.layers.Dropout(0.5),

    # Reduce dim:
    tf.keras.layers.MaxPooling2D(pool_size = [2,2]),
    #tf.keras.layers.Dropout(0.5),

    tf.keras.layers.Flatten(),

    # 10 classes
    tf.keras.layers.Dense(256, activation='relu',
                          use_bias = True, kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Dense(512, activation='relu',
                          use_bias = True, kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Dense(1024, activation='relu',
                          use_bias = True, kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Dense(10, activation='relu',
                          use_bias = True, kernel_initializer = initializer,
                           bias_initializer = initializer)
    ])


    ################################################################################
##
    #                            END OF YOUR CODE
#
    ################################################################################
##
    return model(inputs)

pass

def optimizer_init_fn():
    optimizer = None
    ################################################################################
##
```

```
    # TODO: Construct an optimizer that performs well on CIFAR-10
#
    ################################################################################
##
    optimizer = tf.train.MomentumOptimizer(learning_rate = 5e-4,momentum = 0.9
,use_nesterov = True)

    ################################################################################
##
    #                            END OF YOUR CODE
#
    ################################################################################
##
    return optimizer

device = '/gpu:0'
print_every = 500
num_epochs = 10
train_part34(model_init_fn, optimizer_init_fn, num_epochs)
```

Figure 45. Open Ended Challenge Implementation

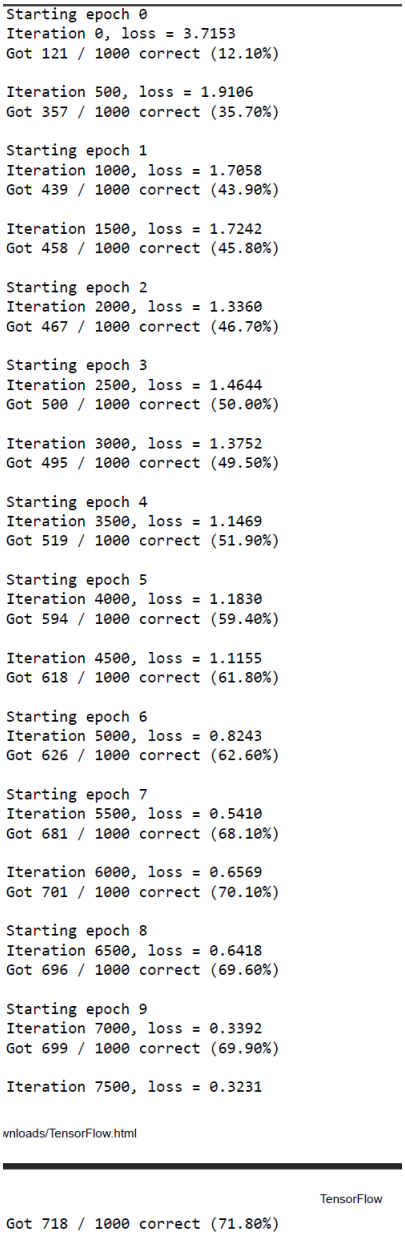


Figure 46. Open Ended Challenge Implementation Training Results

References:

[1] J. Rebello, "Logistic Regression with regularization used to classify hand written digits," 2019. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/42770-logistic-regression-with-regularization-used-to-classify-hand-written-digits?focused=3791937&tab=function. [Accessed 11 December 20019].

Appendix:

Question 1 – MATLAB Code:

```
function question1()
%% Initialize & Explain data:
load('assign3_data1.mat'); clear xForm; clear invXForm;
disp('Question 3 Outputs:');
disp(['Data is of size ', num2str(size(data,1)), ' by ', ...
    num2str(size(data,1)), '.']);
disp(['There are ', num2str(size(data,3)), ' channels.']);
disp(['There are ', num2str(size(data,4)), ' samples.']);
disp('---------------------------');

%% Part A:
% Preprocess to grayscale w/ model: (Y = 0.2126*R + 0.7152*G + 0.0722*B)
R = squeeze(data(:,:,1,:)); G = squeeze(data(:,:,2,:));
B = squeeze(data(:,:,3,:)); ndata = 0.2126*R + 0.7152*G + 0.0722*B;
clear R; clear G; clear B; % Workspace cleanup

% Calculate mean and std to clip & do normalization
mInts = mean(ndata,3); stdInts = std(ndata,[],3);
ndata = ndata - repmat(mInts,1,1,10240); % mean adjustment
clear mInts; % cleanup

% Do a clip mask and reflect it on dataset:
clip = repmat(3*stdInts, 1 ,1, 10240);
negclip = ndata < -clip; posclip = ndata > clip;
ndata = ndata .* ~negclip + ~negclip .*-clip;
ndata = ndata .* ~posclip + ~posclip .*-clip;

% % Old code clips to 0 not to 3-std values
% ndata(ndata < -clip) = -clip;
% ndata(ndata < -clip) = 0; ndata(ndata > clip) = 0;

clear stdInts; clear clip;

% Now rescale:
ndata = rescale(ndata, 0.1, 0.9);

% Display random patches:
plotRandomIms(data, ndata);

%% Part B: - Weight init, uniform Xavier
% Find beta and rho that works well: - I checked some values manually,
% then set the tests around these values as grid search. global optima
% may not reside here more experiments on a wider range is needed
betas = linspace(0.0001,0.01,10); % 10
rhos = linspace(0.1,0.3,5); % 5
[params_b] = generateParamSet(64, 5e-4, betas, rhos , 1);

% Flatten input:
ndata = reshape(ndata, [], 10240); % Reshape to size 256 by 10240

disp('Doing a grid search over beta and rho values:');
% Convert to double (must for doing the calculations)
ndata = double(ndata); data = double(data);

% Test on params:
```

```
[weStar, bestBeta, bestRho, betastar, rhostar] = ...
    partB_experiments(params_b, ndata, betas, rhos);

%% Part C:
paramStar = struct('Lin', 256, 'Lhid', 64, 'lambda',5e-4, ...
    'beta', betas(betastar), 'rho', rhos(rhostar));

% Deconstruct best weights - already calculated:
[w1st, w2st, ~, ~] = unwrapper(weStar, paramStar.Lin, paramStar.Lhid);

% Display weights:
dispWeights(w1st, 8,8);

%% Part D:
% Retrain for different values: - 9 experiments in total
Lhid_exp = linspace(10,100,3);
lambda_exp = linspace(0,0.10e-3,3);
[params_d] = generateParamSet(Lhid_exp, lambda_exp, ...
    bestBeta, bestRho , []);

% Outputs cell array to deal with multi-dim array of different sizes.
[weightsAll] = partD_experiments(params_d, ndata);

% Draw images:
for i = 1:size(params_d,2)
    dispWeights(weightsAll{i},ceil(sqrt(params_d(i).Lhid)), ...
        ceil(sqrt(params_d(i).Lhid)));

    % Beautify plot by adding title: - overrides old title.
    sgtitle(['Hidden layer features with lambda = ', ...
        num2str(params_d(i).lambda), ...
        ' and Lhid = ', num2str(params_d(i).Lhid)]);

end
end


%% Question 1 - Helper Functions:
function [J, Jgrad] = aeCost(we,data,params)
% MATLAB doc suggests Jgrad should be given with nargout
N = size(data,2); a1 = data; % Let a1 = data for ease of reading

% Extract data from given parameters:
Lin = params.Lin; Lhid = params.Lhid; lambda = params.lambda;
beta = params.beta; rho = params.rho;

% Unwrap weights for building the autoencoder
[w1,w2,b1,b2] = unwrapper(we, Lin, Lhid);

% Do autoencoding to extract predictions:
[a2, dz2] = sigmoid(w1 * a1 + repmat(b1,1,N));
[a3, dz3] = sigmoid(w2 * a2 + repmat(b2,1,N));

% Calculate rho_b from avg hidden unit activations:
rho_b = mean(a2,2);

% Calculate KL term:
kl = (rho*log(rho./rho_b) + (1-rho)*log((1-rho)./(1-rho_b)));

% Now do the cost operation: - fminunc takes scalar cost (mean)
J = 1/2 * mean(sum(abs(data-a3).^2,1)) + ...
    lambda/2 * (sum(w1(:).^2) + sum(w2(:).^2)) + ...
```

```
    beta * sum(kl);

% Derivative calculation: KL term - rho has dependency on rho_b, hence a1
% -- w1,b1 terms Tykhonov term - dependency on w1, w2 terms Avg sq. error
% term - dependency on a2 -- w2,w1,b2,b1 terms Calculate backprop for
% squared term, then add respective partial derivs.

% KL derivative:
dKL = (-rho./rho_b + (1-rho)./(1-rho_b));
dKL = repmat(dKL, 1, N); % repeat for all samples

del3 = -(a1-a3) .* dz3;
dw2 = (del3 * a2') ./ N + lambda * w2;
db2 = sum(del3,2) ./ N;

del2 = (w2' * del3  + beta * dKL).* dz2;
dw1 = (del2 * a1') ./ N + lambda * w1;
db1 = sum(del2,2) ./ N;

if nargout > 1 % gradient required
    % Vectorize all weights:
    Jgrad = [dw1(:); dw2(:); db1(:); db2(:)];
end
end
function [w1,w2,b1,b2] = unwrapper(we, Lin, Lhid)
% Unwraps weight vector.
w_leng = Lin*Lhid;
w1 = we(1:w_leng); w1 = reshape(w1, Lhid, Lin);
w2 = we(w_leng+1:2*w_leng); w2 = reshape(w2, Lin, Lhid);
b1 = we(2*w_leng+1:2*w_leng+Lhid);
b2 = we(2*w_leng+Lhid+1:end);
end
function [o, do] = sigmoid(z)
% Calculates sigmoid activation for the neurons, z: activation signal
o = 1./(1+exp(-z));
do = o .* (1-o);
end
function [params] = generateParamSet(Lhids, lambdas, betas, rhos, choice)
if choice == 1
    % For Part B
    c = 1;
    for i = 1: length(betas)
        for j = 1: length(rhos)
            params(c) = struct('Lin', 256, 'Lhid', Lhids, 'lambda',5e-4, ...
                'beta', betas(i), 'rho', rhos(j));
            c = c+1;
        end
    end
else
    % For Part B:
    c = 1;
    for i = 1: length(Lhids)
        for j = 1: length(lambdas)
            params(c) = struct('Lin', 256, 'Lhid', Lhids(i), 'lambda', ...
                lambdas(j),'beta', betas, 'rho', rhos);
            c = c+1;
        end
    end
end
end
%% Flow functions: - Divide and Conquer, ease of readibility and workspace
function plotRandomIms(data, ndata)
```

```
disp('Plotting figures, imshow is slow, this will take some time.');
ranidx = randperm(10240, 200);
figure;
for i = 1:200
    subplot(10,20,i);
    imshow(data(:,:,:,ranidx(i)));
end

% Beautify plot by adding title: -- Works after 2018b, comment if fails.
sgtitle('Random sample patches in RGB format');

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

% Beautify plot by adding title: -- Works after 2018b, comment if fails.
sgtitle('Random sample patches in normalized version');
end
function [weStar, bestBeta, bestRho, betastar, rhost] = ...
    partB_experiments(params, ndata, betas, rhos)
J_min = inf; J_min_idx = -1;  % for min test
for exps = 1: length(params)

    % The weights and biases are symm. in 1 layer case so this is enough:
    wo = sqrt(6/(params(exps).Lhid+params(exps).Lin));

    w1 = -wo + rand(params(exps).Lhid,params(exps).Lin) * 2 * wo;
    w2 = w1'; % tied weights
    b1 = -wo + rand(params(exps).Lhid,1) * 2 * wo;
    b2 = -wo + rand(params(exps).Lin,1) * 2 * wo;

    we = [w1(:); w2(:); b1; b2];

    % Convert everything to double
    we = double(we);

    disp(['Doing experiments, please wait ... ', num2str(exps), '/', ...
        num2str(length(params)), '.']) ;

    % USE fminunc to do gradient descent: - minimum unconstrainted
    opts = optimset('GradObj','on','MaxIter',100); % used in fmincg
    fcn = @(w)aeCost(w,ndata,params(exps));
    [weFinal] = fmincg(fcn, we, opts); % Conj.Gradient Descent
    % I asked this to the course asistant, I was not able to use fminunc as
    % hessian matrix is of 33088 by 33088 size. Question asks for a solver
    % hence I used this one.

    % Now evaluate to see the cost and store it, always keep the minimum
    % cost:
    [J(exps),~] = feval(fcn,weFinal);
    disp(['Cost value = ' , num2str(J(exps)), '.']);
    if J_min > J(exps)
        J_min = J(exps);
        J_min_idx = exps;
        weStar = weFinal;
    end

end

% Find best beta, rho:
```

```
betastar = ceil(J_min_idx / length(rhos));
if mod(J_min_idx,length(rhos)) == 0
   rhost = length(rhos);
else
   rhost = mod(J_min_idx,length(rhos));
end

Jmap = reshape(J, length(betas), length(rhos));

% Display best results:
disp(['Best cost found in index: ', num2str(J_min_idx), ', w/ cost: ', ...
   num2str(J_min), '.']);
disp(['Best beta value is: ', num2str(betas(betastar)), ...
   ', w/ index ' , num2str(betastar) , '.']);
disp(['Best rho value is: ', num2str(rhos(rhost)), ', w/ index ' , ...
   num2str(rhost) , '.']);

bestBeta = betas(betastar);
bestRho = rhos(rhost);

figure; imagesc(Jmap); title(' 2D Experiment Cost Values (Jmap)');
xlabel('rho indices'), ylabel('beta indices'); colorbar;
disp('Note that Jmap can be used to do further refinements on params.');

% ************************************************************************
% NOTE: The solver used in this question is appended below. It is taken
% from the link provided below: (broken into two lines, modified)
% https://www.mathworks.com/matlabcentral/mlc-downloads/downloads/
% submissions/56393/versions/1/previews/mathwork/fmincg.m/index.html
% ************************************************************************
end
function dispWeights(w, xsize, ysize)
% Display first layer connection weights: -- encoder weights
w = reshape(w, [], 16, 16);

% Plot all images:
figure;
for i = 1:size(w,1)
   subplot(xsize,ysize,i);
   imagesc(squeeze(w(i,:,:))); colormap('gray');
end
% Beautify plot by adding title: -- Works after 2018b, comment if fails.
sgtitle('Encoding Weights for the Hidden Layer');
end
function [weightsAll] = partD_experiments(params, ndata)
disp('Doing a grid search over Hidden layer size and lambda values:');
% Test on params:
for exps = 1: length(params)

   % The weights and biases are symm. in 1 layer case so this is enough:
   wo = sqrt(6/(params(exps).Lhid+params(exps).Lin));

   w1 = -wo + rand(params(exps).Lhid,params(exps).Lin) * 2 * wo;
   w2 = w1'; % tied weights
   b1 = -wo + rand(params(exps).Lhid,1) * 2 * wo;
   b2 = -wo + rand(params(exps).Lin,1) * 2 * wo;

   we = [w1(:); w2(:); b1; b2];

   % Convert everything to double
   we = double(we);
```

```
    disp(['Doing experiments, please wait ... ', num2str(exps), '/', ...
        num2str(length(params)), '.']) ;

    % USE fminunc to do gradient descent: - minimum unconstrainted
    opts = optimset('GradObj','on','MaxIter',100); % used in fmincg
    fcn = @(w)aeCost(w,ndata,params(exps));
    [weFinal] = fmincg(fcn, we, opts); % Conj.Gradient Descent

    % Now evaluate to see the cost and store it, always keep the minimum
    % cost:
    [J(exps),~] = feval(fcn,weFinal);

    disp(['Cost value = ' , num2str(J(exps)), '.']);
    % I asked this to the course asistant, I was not able to use fminunc as
    % hessian matrix is of 33088 by 33088 size. Question asks for a solver
    % hence I used this one.

    % We just want to print out hidden layer features/weights so unwrap and
    % send only that information for display:
    % Deconstruct best weights - already calculated:
    [w1st, w2st, ~, ~] = unwrapper(weFinal, params(exps).Lin, params(exps).Lhid);


    weightsAll{exps} = w1st;

    %   % Deconstruct best weights - already calculated:
    %   [w1st, w2st, ~, ~] = unwrapper(weFinal, params(exps).Lin, ...
    %       params(exps).Lhid);
    %
    %   % Display first layer connection weights: -- encoder weights
    %   w1_disp = reshape(w1st, params(exps).Lhid, 16, 16);
    %
    %   % Plot all images:
    %   figure;
    %   for i = 1:size(w1_disp,1)
    %       subplot(10,10,i);
    %       imagesc(squeeze(w1_disp(i,:,:))); colormap('gray');
    %   end
    %
    %   % Beautify plot by adding title: - Works after 2018b, comment if fails.
    %   sgtitle(['Encoding Weights for the Hidden Layer Test ', num2str(i)]);

end
end

%% External Code, references given:
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****pas coherentes entres elles (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 reeval 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)
%**douille


while i < abs(length)                            % while not finished
   i = i + (length>0);                           % count iterations?!
   % fprintf('test verif minim :  %f\n',i);
   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 acc 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 clos to lim?
      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 pnt from before failed 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
end
%fprintf('\n');
%fprintf('%s %4i | Cost: %4.6e\r', S, i, f1);
end
```

Question 2 – MATLAB Code:

It is an empty function indicating this question has no MATLAB counterpart.

```
function question2()
disp('This question is implemented in Phyton by using given environment.');
disp('No output is available.');
end
```

Assignment 3 – Main Code:

```
% ********************************************************************************
% EEE 443 - Neural Networks - Assignment 3
% Hasan Emre Erdemoglu - 21401462
% Due: Dec, 15 2019; 23.55 PM
% ********************************************************************************
function hasanemre_erdemoglu_21401462_hw3(question)
clc
close all

switch question
   case '1'
      disp('Question 1:')
      question1();
   case '2'
      disp('Question 2:')
      % This question normally does not have a MATLAB counterpart.
      question2();
end

end
```

Question 2 - PDF Appendix:

The appendix for both parts A and B can be found in their respective order next page.

Convolutional Networks

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

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

In [1]:
```
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerica
l_gradient
from cs231n.layers import *

from cs231n.fast_layers import *
from cs231n.solver import Solver

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

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

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

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

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

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

Convolution: Naive forward pass

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

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

You can test your implementation by running the following:

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

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

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

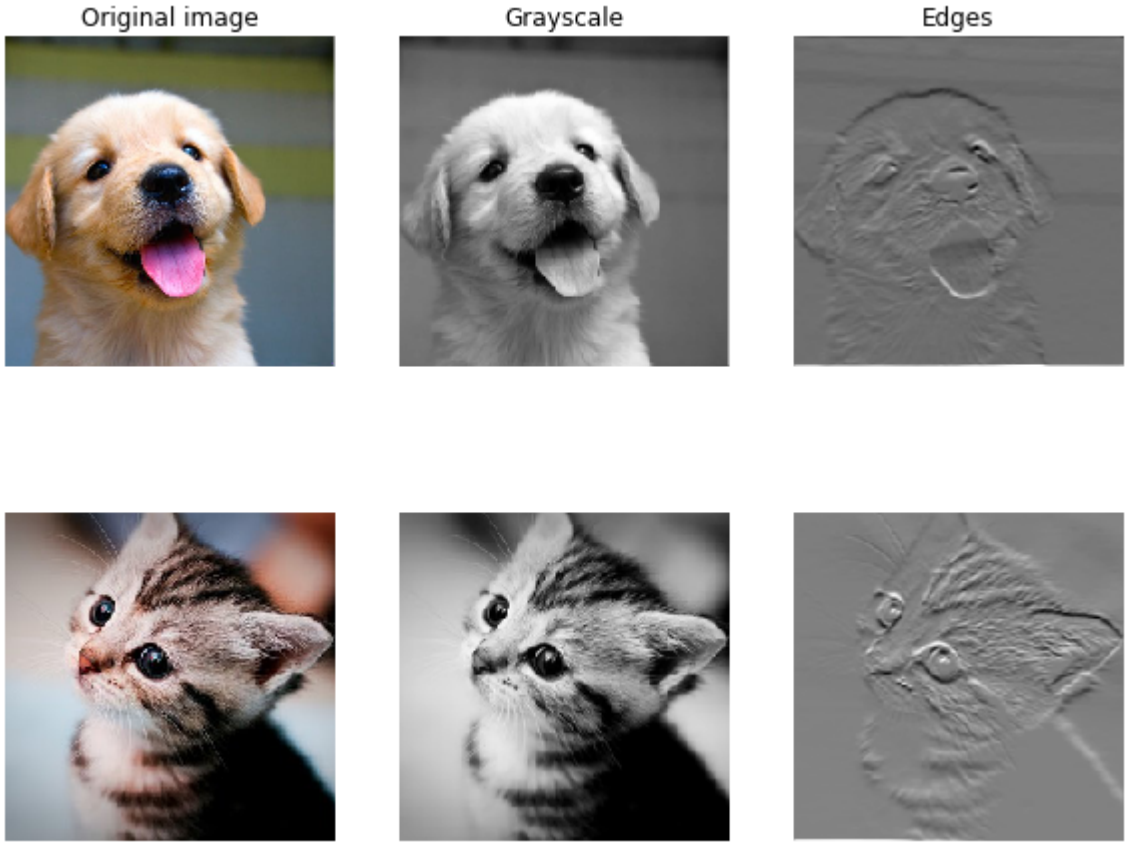
```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

Aside: Image processing via convolutions

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

In [4]:
```
#from scipy.misc import imread, imresize # imread and imresize deprecated use
 below -- emre
from imageio import imread
import skimage

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, :, :, :] = skimage.transform.resize(puppy, (img_size, img_size)).transpos
e((2, 0, 1))
x[1, :, :, :] = skimage.transform.resize(kitten_cropped, (img_size, img_size))
.transpose((2, 0, 1))

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

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

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

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

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

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

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

```
plt.subplot(2, 3, 4)
imshow_noax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_noax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```



Convolution: Naive backward pass

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

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

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

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

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

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

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

Max-Pooling: Naive forward

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

Check your implementation by running the following:

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

out, _ = max_pool_forward_naive(x, pool_param)

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

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

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

Max-Pooling: Naive backward

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

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

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

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

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

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

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

Fast layers

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

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

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

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

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

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

In [8]:
```
# Rel errors should be around e-9 or less
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(231)
x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

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

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

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

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

```
Testing conv_forward_fast:
Naive: 4.382130s
Fast: 0.015961s
Speedup: 274.557600x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 6.781001s
Fast: 0.013001s
Speedup: 521.567182x
dx difference:  1.949764775345631e-11
dw difference:  4.4985195578905695e-13
db difference:  0.0
```

In [9]:
```
# Relative errors should be close to 0.0
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(231)
x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

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

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

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

```
Testing pool_forward_fast:
Naive: 0.145034s
fast: 0.003964s
speedup: 36.586155x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.443965s
fast: 0.010035s
speedup: 44.243561x
dx difference:  0.0
```

Convolutional "sandwich" layers

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

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

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

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

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

```
Testing conv_relu_pool
dx error:  6.514336569263308e-09
dw error:  1.490843753539445e-08
db error:  2.037390356217257e-09
```

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

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

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

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

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

Three-layer ConvNet

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

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

Sanity check loss

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

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

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

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

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

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

Gradient check

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

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

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
                          dtype=np.float64)
loss, grads = model.loss(X, y)
# Errors should be small, but correct implementations may have
# relative errors up to the order of e-2
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verb
ose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,
grads[param_name])))
```

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

Overfit small data

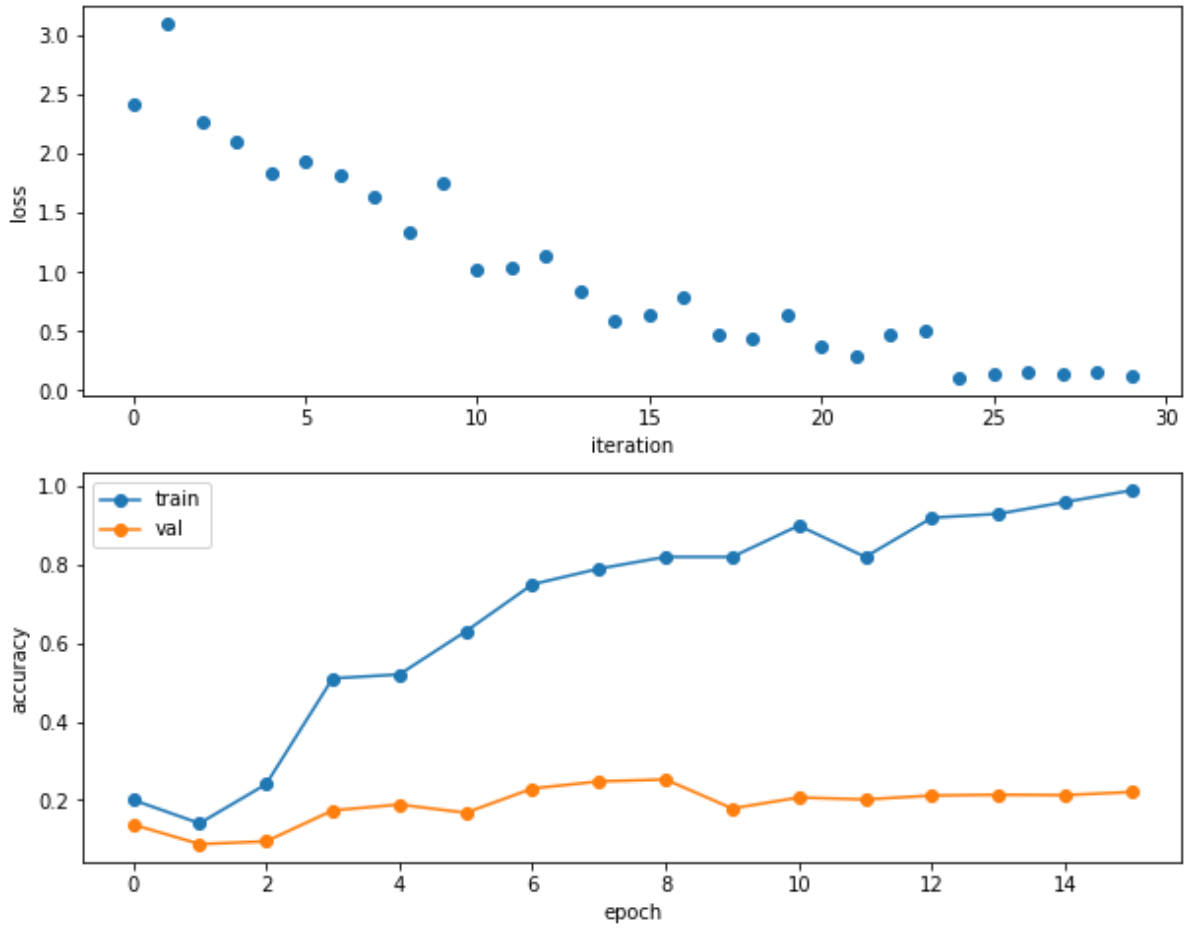
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

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

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

model = ThreeLayerConvNet(weight_scale=1e-2)

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

```
(Iteration 1 / 30) loss: 2.414060
(Epoch 0 / 15) train acc: 0.200000; val_acc: 0.137000
(Iteration 2 / 30) loss: 3.102925
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.270330
(Iteration 4 / 30) loss: 2.096705
(Epoch 2 / 15) train acc: 0.240000; val_acc: 0.094000
(Iteration 5 / 30) loss: 1.838880
(Iteration 6 / 30) loss: 1.934188
(Epoch 3 / 15) train acc: 0.510000; val_acc: 0.173000
(Iteration 7 / 30) loss: 1.827912
(Iteration 8 / 30) loss: 1.639574
(Epoch 4 / 15) train acc: 0.520000; val_acc: 0.188000
(Iteration 9 / 30) loss: 1.330082
(Iteration 10 / 30) loss: 1.756115
(Epoch 5 / 15) train acc: 0.630000; val_acc: 0.167000
(Iteration 11 / 30) loss: 1.024162
(Iteration 12 / 30) loss: 1.041826
(Epoch 6 / 15) train acc: 0.750000; val_acc: 0.229000
(Iteration 13 / 30) loss: 1.142777
(Iteration 14 / 30) loss: 0.835706
(Epoch 7 / 15) train acc: 0.790000; val_acc: 0.247000
(Iteration 15 / 30) loss: 0.587786
(Iteration 16 / 30) loss: 0.645509
(Epoch 8 / 15) train acc: 0.820000; val_acc: 0.252000
(Iteration 17 / 30) loss: 0.786844
(Iteration 18 / 30) loss: 0.467054
(Epoch 9 / 15) train acc: 0.820000; val_acc: 0.178000
(Iteration 19 / 30) loss: 0.429880
(Iteration 20 / 30) loss: 0.635498
(Epoch 10 / 15) train acc: 0.900000; val_acc: 0.206000
(Iteration 21 / 30) loss: 0.365807
(Iteration 22 / 30) loss: 0.284220
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.201000
(Iteration 23 / 30) loss: 0.469343
(Iteration 24 / 30) loss: 0.509369
(Epoch 12 / 15) train acc: 0.920000; val_acc: 0.211000
(Iteration 25 / 30) loss: 0.111638
(Iteration 26 / 30) loss: 0.145388
(Epoch 13 / 15) train acc: 0.930000; val_acc: 0.213000
(Iteration 27 / 30) loss: 0.155575
(Iteration 28 / 30) loss: 0.143398
(Epoch 14 / 15) train acc: 0.960000; val_acc: 0.212000
(Iteration 29 / 30) loss: 0.158160
(Iteration 30 / 30) loss: 0.118934
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.220000
```

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

In [15]:
```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

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



Train the net

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

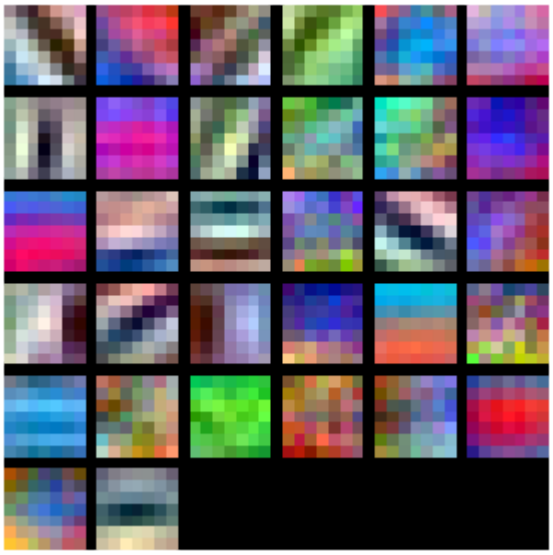
In [16]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

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

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

Visualize Filters

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

In [18]:
```
from cs231n.vis_utils import visualize_grid

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



Spatial Batch Normalization

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

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

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

[3] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015. (https://arxiv.org/abs/1502.03167)

Spatial batch normalization: forward

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

In [19]:
```
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 [20]:
```
np.random.seed(231)
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

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

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

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

Spatial batch normalization: backward

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

In [21]:
```
np.random.seed(231)
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

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

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

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

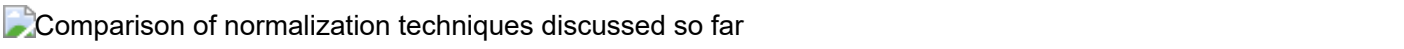
```
dx error:  3.083846820796372e-07
dgamma error:  7.09738489671469e-12
dbeta error:  3.275608725278405e-12
```

Group Normalization

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

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

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


Comparison of normalization techniques discussed so far

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

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

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

[4] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21. (https://arxiv.org/pdf/1607.06450.pdf)

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

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

Group normalization: forward

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

In [22]:
```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

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

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

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

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

Spatial group normalization: backward

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

In [23]:
```
np.random.seed(231)
N, C, H, W = 2, 6, 4, 5
G = 2
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

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

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

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

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

In [ ]:

What's this TensorFlow business?

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

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

What is it?

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

Why?

- Our code will now run on GPUs! Much faster training. Writing your own modules to run on GPUs is beyond the scope of this class, unfortunately.
- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

How will I learn TensorFlow?

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

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

Table of Contents

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

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

Here is a table of comparison:

API	Flexibility	Convenience
Barebone	High	Low
`tf.keras.Model`	High	Medium
`tf.keras.Sequential`	Low	High

Part I: Preparation

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

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

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

In [3]:
```
import os
import tensorflow as tf
import numpy as np
import math
import timeit
import matplotlib.pyplot as plt

print(tf.__version__)
%matplotlib inline
```

```
1.14.0
```

In [4]:
```
def load_cifar10(num_training=49000, num_validation=1000, num_test=10000):
    """
    Fetch the CIFAR-10 dataset from the web and perform preprocessing to prepa
re
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 dataset and use appropriate data types and shapes
    cifar10 = tf.keras.datasets.cifar10.load_data()
    (X_train, y_train), (X_test, y_test) = cifar10
    X_train = np.asarray(X_train, dtype=np.float32)
    y_train = np.asarray(y_train, dtype=np.int32).flatten()
    X_test = np.asarray(X_test, dtype=np.float32)
    y_test = np.asarray(y_test, dtype=np.int32).flatten()

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

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

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


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

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

Preparation: Dataset object

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

In [5]:
```
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 [6]:
```
# We can iterate through a dataset like this:
for t, (x, y) in enumerate(train_dset):
    print(t, x.shape, y.shape)
    if t > 5: break
```

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

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

In [7]:
```
# Set up some global variables
USE_GPU = True

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:  /device:GPU:0
```

Part II: Barebone TensorFlow

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

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

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

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

TensorFlow warmup: Flatten Function

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

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

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

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

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

In [8]:
```
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 [9]:
```
def test_flatten():
    # Clear the current TensorFlow graph.
    tf.reset_default_graph()

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

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

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

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

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

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

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

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

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

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

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

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

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

Barebones TensorFlow: Two-Layer Network

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

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

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

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

In [10]:
```
def two_layer_fc(x, params):
    """
    A fully-connected neural network; the architecture is:
    fully-connected layer -> ReLU -> fully connected layer.
    Note that we only need to define the forward pass here; TensorFlow will ta
ke
    care of computing the gradients for us.

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

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

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

In [11]:
```
def two_layer_fc_test():
    # TensorFlow's default computational graph is essentially a hidden global
    # variable. To avoid adding to this default graph when you rerun this cel
l,
    # we clear the default graph before constructing the graph we care about.
    tf.reset_default_graph()
    hidden_layer_size = 42

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

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

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

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

two_layer_fc_test()
```

```
(64, 10)
```

Barebones TensorFlow: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet` which will perform the forward pass of a three-layer convolutional network. The network should have the following architecture:

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

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

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

In [12]:
```
def three_layer_convnet(x, params):
    """
    A three-layer convolutional network with the architecture described above.

    Inputs:
    - x: A TensorFlow Tensor of shape (N, H, W, 3) giving a minibatch of image
s
    - params: A list of TensorFlow Tensors giving the weights and biases for t
he
      network; should contain the following:
      - conv_w1: TensorFlow Tensor of shape (KH1, KW1, 3, channel_1) giving
        weights for the first convolutional layer.
      - conv_b1: TensorFlow Tensor of shape (channel_1,) giving biases for the
        first convolutional layer.
      - conv_w2: TensorFlow Tensor of shape (KH2, KW2, channel_1, channel_2)
        giving weights for the second convolutional layer
      - conv_b2: TensorFlow Tensor of shape (channel_2,) giving biases for the
        second convolutional layer.
      - fc_w: TensorFlow Tensor giving weights for the fully-connected layer.
        Can you figure out what the shape should be?
      - fc_b: TensorFlow Tensor giving biases for the fully-connected layer.
        Can you figure out what the shape should be?
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None
    ################################################################################
##
    # TODO: Implement the forward pass for the three-layer ConvNet.
#
    ################################################################################
##
    layer1_conv = tf.nn.conv2d(input = x,filter = conv_w1,
                               strides = [1,1,1,1], padding = 'SAME', name =
'conv1') + conv_b1
    layer1_conv_relu = tf.nn.relu(layer1_conv)

    layer2_conv = tf.nn.conv2d(input = layer1_conv_relu,filter = conv_w2,
                               strides = [1,1,1,1],padding = 'SAME' ,name = 'c
onv2') + conv_b2
    layer2_conv_relu = tf.nn.relu(layer2_conv)

    layer3 = flatten(layer2_conv_relu)

    scores = tf.matmul(layer3,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 [13]:
```
def three_layer_convnet_test():
    tf.reset_default_graph()

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

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

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

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

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

Barebones TensorFlow: Training Step

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

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

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

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

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

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

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

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

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

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

Barebones TensorFlow: Training Loop

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

In [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 passe
s,
    # and weight updates.
    with tf.device(device):
        # Set up placeholders for the data and labels
        x = tf.placeholder(tf.float32, [None, 32, 32, 3])
        y = tf.placeholder(tf.int32, [None])
        params = init_fn()           # Initialize the model parameters
        scores = model_fn(x, params) # Forward pass of the model
        loss = training_step(scores, y, params, learning_rate)

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

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

Barebones TensorFlow: Check Accuracy

When training the model we will use the following function to check the accuracy of our model on the training or validation sets. Note that this function accepts a TensorFlow Session object as one of its arguments; this is needed since the function must actually run the computational graph many times on the data that it loads from the dataset `dset`.

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

In [16]:
```
def check_accuracy(sess, dset, x, scores, is_training=None):
    """
    Check accuracy on a classification model.

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

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

Barebones TensorFlow: Initialization

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

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

In [17]:
```
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 [18]:
```
def two_layer_fc_init():
    """
    Initialize the weights of a two-layer network, for use with the
    two_layer_network function defined above.

    Inputs: None

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

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

```
Iteration 0, loss = 3.1610
Got 108 / 1000 correct (10.80%)
Iteration 100, loss = 1.8886
Got 375 / 1000 correct (37.50%)
Iteration 200, loss = 1.5118
Got 371 / 1000 correct (37.10%)
Iteration 300, loss = 1.7202
Got 378 / 1000 correct (37.80%)
Iteration 400, loss = 1.7810
Got 408 / 1000 correct (40.80%)
Iteration 500, loss = 1.7605
Got 434 / 1000 correct (43.40%)
Iteration 600, loss = 1.9183
Got 421 / 1000 correct (42.10%)
Iteration 700, loss = 1.9711
Got 448 / 1000 correct (44.80%)
```

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 [19]:
```
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 = 3.0160
Got 95 / 1000 correct (9.50%)
Iteration 100, loss = 1.9559
Got 300 / 1000 correct (30.00%)
Iteration 200, loss = 1.8292
Got 309 / 1000 correct (30.90%)
Iteration 300, loss = 2.0164
Got 296 / 1000 correct (29.60%)
Iteration 400, loss = 1.9395
Got 347 / 1000 correct (34.70%)
Iteration 500, loss = 1.9531
Got 341 / 1000 correct (34.10%)
Iteration 600, loss = 1.9506
Got 366 / 1000 correct (36.60%)
Iteration 700, loss = 1.6701
Got 355 / 1000 correct (35.50%)
```

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)](https://www.tensorflow.org/programmers_guide/faq#tensor_shapes).

In [20]:
```
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) # this is deprecated so I changed it - EMRE
        x = tf.compat.v1.layers.Flatten()(x) # https://stackoverflow.com/quest
ions/53153790/tensor-object-has-no-attribute-lower
        # First instantiate then initialize
        x = self.fc1(x)
        x = self.fc2(x)
        return x


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

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

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

test_TwoLayerFC()
```

```
(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 [21]:
```
def two_layer_fc_functional(inputs, hidden_size, num_classes):
    initializer = tf.variance_scaling_initializer(scale=2.0)
    flattened_inputs = tf.layers.flatten(inputs)
    fc1_output = tf.layers.dense(flattened_inputs, hidden_size, activation=tf.
nn.relu,
                                 kernel_initializer=initializer)
    scores = tf.layers.dense(fc1_output, num_classes,
                             kernel_initializer=initializer)
    return scores

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

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

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

test_two_layer_fc_functional()
```

```
WARNING:tensorflow:From <ipython-input-21-e2eb30b3f3fa>:3: flatten (from tens
orflow.python.layers.core) is deprecated and will be removed in a future vers
ion.
Instructions for updating:
Use keras.layers.flatten instead.
WARNING:tensorflow:From <ipython-input-21-e2eb30b3f3fa>:5: dense (from tensor
flow.python.layers.core) is deprecated and will be removed in a future versio
n.
Instructions for updating:
Use keras.layers.dense instead.
(64, 10)
```

Keras Model API: Three-Layer ConvNet

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

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

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

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

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

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

In [22]:
```
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.
#
        ########################################################################
##
        # 5x5 kernels, relu non-linearity
        var_init = tf.variance_scaling_initializer(scale=2.0)
        self.conv1 = tf.layers.Conv2D(filters = channel_1,kernel_size = [5,5],
                                      strides = [1,1],padding = 'SAME',activat
ion = tf.nn.relu,
                                     use_bias = True,kernel_initializer = var_
init,
                                      bias_initializer = var_init,name = 'conv
1')

        # 3x3 kernels, relu non-linearity
        self.conv2 = tf.layers.Conv2D(filters = channel_2,kernel_size = [3,3],
                                      strides = [1,1],padding = 'SAME',activat
ion = tf.nn.relu,
                                     use_bias = True,kernel_initializer = var_
init,
                                      bias_initializer = var_init,name = 'conv
2')
        # Fully connected layer
        self.fc = tf.layers.Dense(units = num_classes,use_bias = True,
                                  kernel_initializer = var_init,bias_initializ
er = var_init,
                                  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_c1 = self.conv1(x)
        x_c2 = self.conv2(x_c1)
        x_c2_flat = tf.layers.flatten(x_c2)
        scores = self.fc(x_c2_flat)
        ########################################################################
```

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

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

In [23]:
```
def test_ThreeLayerConvNet():
    tf.reset_default_graph()

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

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

test_ThreeLayerConvNet()
```

```
(64, 10)
```

Keras Model API: Training Loop

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

In [24]:
```
def train_part34(model_init_fn, optimizer_init_fn, num_epochs=1):
    """
    Simple training loop for use with models defined using tf.keras. It trains
    a model for one epoch on the CIFAR-10 training set and periodically checks
    accuracy on the CIFAR-10 validation set.

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

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

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

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

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

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

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

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

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

Keras Model API: Train a Two-Layer Network

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

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

In [25]:
```
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 = 2.8361
Got 141 / 1000 correct (14.10%)

Iteration 100, loss = 1.8689
Got 393 / 1000 correct (39.30%)

Iteration 200, loss = 1.4486
Got 388 / 1000 correct (38.80%)

Iteration 300, loss = 1.7736
Got 377 / 1000 correct (37.70%)

Iteration 400, loss = 1.8063
Got 442 / 1000 correct (44.20%)

Iteration 500, loss = 1.9122
Got 432 / 1000 correct (43.20%)

Iteration 600, loss = 1.8056
Got 436 / 1000 correct (43.60%)

Iteration 700, loss = 1.9277
Got 444 / 1000 correct (44.40%)
```

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

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

In [26]:
```
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 = 2.9702
Got 117 / 1000 correct (11.70%)

Iteration 100, loss = 1.9470
Got 394 / 1000 correct (39.40%)

Iteration 200, loss = 1.4413
Got 379 / 1000 correct (37.90%)

Iteration 300, loss = 1.8021
Got 368 / 1000 correct (36.80%)

Iteration 400, loss = 1.8578
Got 426 / 1000 correct (42.60%)

Iteration 500, loss = 1.7655
Got 429 / 1000 correct (42.90%)

Iteration 600, loss = 1.7955
Got 419 / 1000 correct (41.90%)

Iteration 700, loss = 1.9614
Got 444 / 1000 correct (44.40%)
```

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 [27]:
```
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,moment
um = 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 = 3.0034
Got 110 / 1000 correct (11.00%)

Iteration 100, loss = 1.4748
Got 453 / 1000 correct (45.30%)

Iteration 200, loss = 1.3288
Got 480 / 1000 correct (48.00%)

Iteration 300, loss = 1.4024
Got 495 / 1000 correct (49.50%)

Iteration 400, loss = 1.1305
Got 522 / 1000 correct (52.20%)

Iteration 500, loss = 1.3949
Got 554 / 1000 correct (55.40%)

Iteration 600, loss = 1.3448
Got 550 / 1000 correct (55.00%)

Iteration 700, loss = 1.2514
Got 552 / 1000 correct (55.20%)
```

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 [28]:
```
learning_rate = 1e-2

# The tf.layers used here are deprecated so I changed them to their modern cou
nterparts.
# tf.layers.Flatten --> tf.keras.layers.Flatten

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 = 2.9567
Got 117 / 1000 correct (11.70%)

Iteration 100, loss = 1.8102
Got 386 / 1000 correct (38.60%)

Iteration 200, loss = 1.4827
Got 397 / 1000 correct (39.70%)

Iteration 300, loss = 1.8948
Got 374 / 1000 correct (37.40%)

Iteration 400, loss = 2.0211
Got 425 / 1000 correct (42.50%)

Iteration 500, loss = 1.7565
Got 434 / 1000 correct (43.40%)

Iteration 600, loss = 1.9392
Got 437 / 1000 correct (43.70%)

Iteration 700, loss = 1.9409
Got 451 / 1000 correct (45.10%)
```

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 [29]:
```
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.relu,
                               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.relu,
                               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 = initializer,
                               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 = 5e-4,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 = 2.9914
Got 101 / 1000 correct (10.10%)

Iteration 100, loss = 1.7236
Got 413 / 1000 correct (41.30%)

Iteration 200, loss = 1.4269
Got 468 / 1000 correct (46.80%)

Iteration 300, loss = 1.3530
Got 480 / 1000 correct (48.00%)

Iteration 400, loss = 1.4411
Got 504 / 1000 correct (50.40%)

Iteration 500, loss = 1.5887
Got 506 / 1000 correct (50.60%)

Iteration 600, loss = 1.5389
Got 524 / 1000 correct (52.40%)

Iteration 700, loss = 1.4681
Got 542 / 1000 correct (54.20%)
```

Part V: CIFAR-10 open-ended challenge

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

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

Describe what you did at the end of the notebook.

Some things you can try:

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

WARNING: Batch Normalization / Dropout

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

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

Tips for training

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

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

- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

Going above and beyond

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

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.
- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.
- Model ensembles
- Data augmentation
- New Architectures
  - ResNets (https://arxiv.org/abs/1512.03385) where the input from the previous layer is added to the output.
  - DenseNets (https://arxiv.org/abs/1608.06993) where inputs into previous layers are concatenated together.
  - This blog has an in-depth overview (https://chatbotslife.com/resnets-highwaynets-and-densenets-oh-my-9bb15918ee32)

Have fun and happy training!

TODO: Tell us what you did

In [38]:
```
def model_init_fn(inputs, is_training):
    model = None
    ############################################################################
##
    # TODO: Construct a model that performs well on CIFAR-10
#
    ############################################################################
##
    initializer = tf.variance_scaling_initializer(scale=2.0)

    model = tf.keras.models.Sequential([
    # Layer 1-3:
    tf.keras.layers.Conv2D(input_shape = (32,32,3),filters =64,kernel_size = [
3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 64,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 64,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    # Reduce dim:
    tf.keras.layers.MaxPooling2D(pool_size = [2,2]),
    #tf.keras.layers.Dropout(0.5),

    # Layer 4-6:
    tf.keras.layers.Conv2D(filters = 128,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 128,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 128,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    # Reduce dim:
    tf.keras.layers.MaxPooling2D(pool_size = [2,2]),
    #tf.keras.layers.Dropout(0.4),

    # Layer 7-9:
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
                           padding = 'SAME',activation = 'relu',
                           use_bias = True,kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
```

```
                            padding = 'SAME',activation = 'relu',
                            use_bias = True,kernel_initializer = initializer,
                            bias_initializer = initializer),
    # Layer 10-12:
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
                            padding = 'SAME',activation = 'relu',
                            use_bias = True,kernel_initializer = initializer,
                            bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
                            padding = 'SAME',activation = 'relu',
                            use_bias = True,kernel_initializer = initializer,
                            bias_initializer = initializer),
    tf.keras.layers.Conv2D(filters = 256,kernel_size = [3,3],
                            padding = 'SAME',activation = 'relu',
                            use_bias = True,kernel_initializer = initializer,
                            bias_initializer = initializer),

    # Reduce dim:
    tf.keras.layers.MaxPooling2D(pool_size = [2,2]),
    #tf.keras.layers.Dropout(0.5),

    # Reduce dim:
    tf.keras.layers.MaxPooling2D(pool_size = [2,2]),
    #tf.keras.layers.Dropout(0.5),

    tf.keras.layers.Flatten(),

    # 10 classes
    tf.keras.layers.Dense(256, activation='relu',
                          use_bias = True, kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Dense(512, activation='relu',
                          use_bias = True, kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Dense(1024, activation='relu',
                          use_bias = True, kernel_initializer = initializer,
                           bias_initializer = initializer),
    tf.keras.layers.Dense(10, activation='relu',
                          use_bias = True, kernel_initializer = initializer,
                           bias_initializer = initializer)
    ])


    ################################################################################
##
    #                               END OF YOUR CODE
#
    ################################################################################
##
    return model(inputs)

pass

def optimizer_init_fn():
    optimizer = None
    ################################################################################
##
```

```
    # TODO: Construct an optimizer that performs well on CIFAR-10
#
    ###############################################################################
##
    optimizer = tf.train.MomentumOptimizer(learning_rate = 5e-4,momentum = 0.9
,use_nesterov = True)

    ###############################################################################
##
    #                                END OF YOUR CODE
#
    ###############################################################################
##
    return optimizer

device = '/gpu:0'
print_every = 500
num_epochs = 10
train_part34(model_init_fn, optimizer_init_fn, num_epochs)
```

```
Starting epoch 0
Iteration 0, loss = 3.7153
Got 121 / 1000 correct (12.10%)

Iteration 500, loss = 1.9106
Got 357 / 1000 correct (35.70%)

Starting epoch 1
Iteration 1000, loss = 1.7058
Got 439 / 1000 correct (43.90%)

Iteration 1500, loss = 1.7242
Got 458 / 1000 correct (45.80%)

Starting epoch 2
Iteration 2000, loss = 1.3360
Got 467 / 1000 correct (46.70%)

Starting epoch 3
Iteration 2500, loss = 1.4644
Got 500 / 1000 correct (50.00%)

Iteration 3000, loss = 1.3752
Got 495 / 1000 correct (49.50%)

Starting epoch 4
Iteration 3500, loss = 1.1469
Got 519 / 1000 correct (51.90%)

Starting epoch 5
Iteration 4000, loss = 1.1830
Got 594 / 1000 correct (59.40%)

Iteration 4500, loss = 1.1155
Got 618 / 1000 correct (61.80%)

Starting epoch 6
Iteration 5000, loss = 0.8243
Got 626 / 1000 correct (62.60%)

Starting epoch 7
Iteration 5500, loss = 0.5410
Got 681 / 1000 correct (68.10%)

Iteration 6000, loss = 0.6569
Got 701 / 1000 correct (70.10%)

Starting epoch 8
Iteration 6500, loss = 0.6418
Got 696 / 1000 correct (69.60%)

Starting epoch 9
Iteration 7000, loss = 0.3392
Got 699 / 1000 correct (69.90%)

Iteration 7500, loss = 0.3231
```

```
Got 718 / 1000 correct (71.80%)
```

In [ ]:

In [ ]: