Abdul Basit Anees
21600659
EEE-443

Neural Networks

Assignment 3

Question 1

A

The images from the file were loaded and grayscale images were generated using the model:
Y = 0:2126 * R + 0:7152 * G + 0:0722 * B

The mean was removed and the data was clipped at +-3 standard deviations after which it was mapped to the range [0.1 0.9]. Following are the 200 RGB images with their corresponding normalized grayscale images.

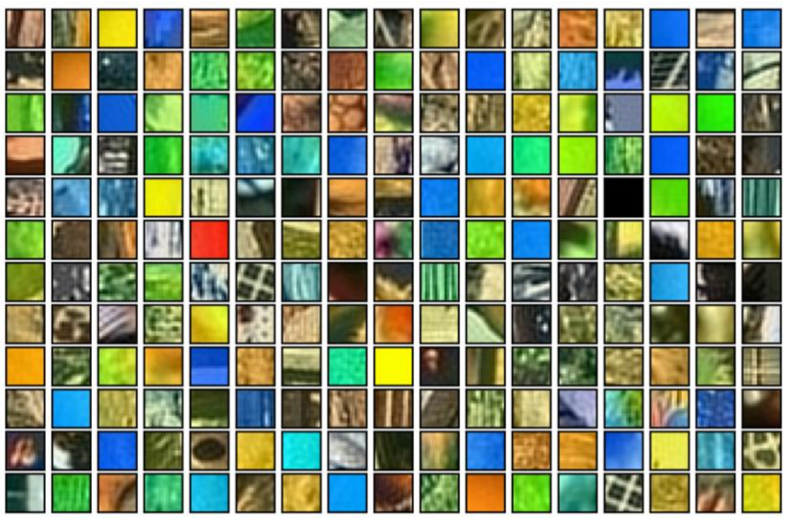


Figure 1: RGB images

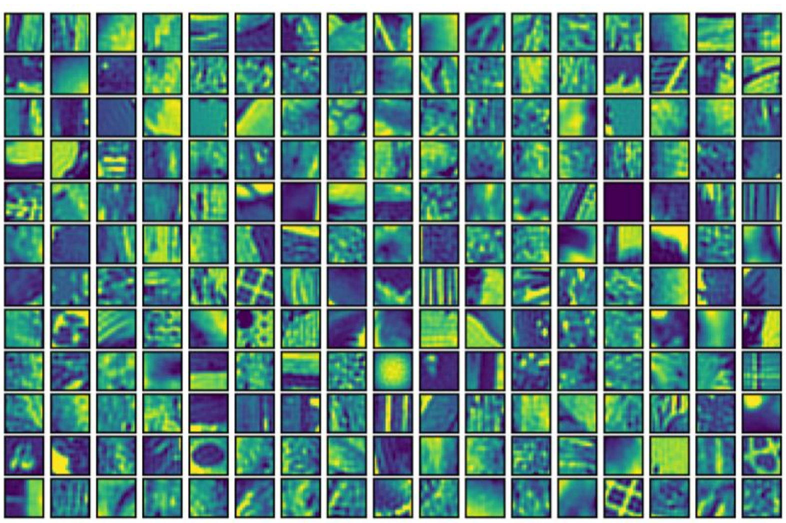


Figure 2: Normalized grayscale images

Abdul Basit Anees
21600659
EEE-443

First of all, converting to grayscale shrinks our data size by 3. Moreover, with proper scaling of each channel, all the channels represent equal amount of information and patterns in the image are much more visible.

B and C

The loss function is defined as:

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

$$J = J_1 + J_2 + KL$$

The partial derivatives of the loss function of the KL term are as follows

$\boxed{\frac{\mathrm{dKL}}{\mathrm{dw}_2} = 0}$ Since hidden layer activations don't depend on them.

$$\mathrm{KL} = \beta . \sum \mathrm{KL}(\delta|\hat{\delta}_j)$$

$$\frac{\mathrm{dKL}(\delta|\hat{\delta}_j)}{\mathrm{dw}_1} = \frac{\mathrm{dKL}(\delta|\hat{\delta}_j)}{\mathrm{d}\hat{\delta}_j} . \frac{\mathrm{d}\hat{\delta}_j}{\mathrm{dw}_1}$$

$$\frac{\mathrm{dKL}(\delta|\hat{\delta}_j)}{\mathrm{d}\hat{\delta}_j} = \frac{1-\delta}{\hat{\delta}_j} - \frac{\delta}{\hat{\delta}_j}$$

$$\hat{\delta}_j = average\ of\ the\ outputs\ for\ all\ inputs\ in\ the\ batch\ for\ the\ jth\ hidden\ layer\ neuron$$

Let $\underline{\hat{\delta}} = vector\ of\ all\ \hat{\delta}_j$

$$\frac{\mathrm{dKL}}{\mathrm{d}\underline{\hat{\delta}}} = Vector\ of\ \frac{\mathrm{dKL}(\delta|\hat{\delta}_j)}{\mathrm{d}\hat{\delta}_j} \text{for all j}$$

$$\frac{\mathrm{dKL}}{\mathrm{d}\underline{\hat{\delta}}} = \frac{1-\delta}{\underline{\hat{\delta}}} - \frac{\delta}{\underline{\hat{\delta}}}$$

$$\underline{\hat{\delta}} = \frac{\sum \varphi(x^i @ \mathrm{w}_1 + b_1)}{batch\ size}$$

$$\frac{\mathrm{d}\underline{\hat{\delta}}}{\mathrm{dw}_1} = \frac{\sum x^i @\ \varphi'(x^i @ \mathrm{w}_1 + b_1)}{batch\ size}$$

Let $\frac{\mathrm{d}\underline{\hat{\delta}}}{\mathrm{dw}_1}$(k) denote the kth row of $\frac{\mathrm{d}\underline{\hat{\delta}}}{\mathrm{dw}_1}$

Let j denote the jth row of w_1

$$\frac{\mathrm{d}\hat{\delta}_j}{\mathrm{dw}_1} = 0\ if\ j\ \neq k$$

$$\frac{\mathrm{d}\hat{\delta}_j}{\mathrm{dw}_1} = \frac{\mathrm{d}\underline{\hat{\delta}}}{\mathrm{dw}_1}(\mathrm{k})\ \ if\ j = k$$

Abdul Basit Anees
21600659
EEE-443

$$\frac{\mathrm{d}\hat{\delta}_j}{\mathrm{dw}_1} = 0 \; everywhere\ except\ the\ jth\ row\ where\ \frac{\mathrm{d}\hat{\delta}_j}{\mathrm{dw}_1} = \frac{\mathrm{d}\underline{\hat{\delta}}}{\mathrm{dw}_1}(\mathrm{j})$$

$$\sum \frac{\mathrm{d}\hat{\delta}_j}{\mathrm{dw}_1} = \frac{\mathrm{d}\underline{\hat{\delta}}}{\mathrm{dw}_1}\ since\ all\ \frac{\mathrm{d}\hat{\delta}_j}{\mathrm{dw}_1}\ are\ linearly\ independent$$

$$\boxed{\sum \frac{\mathrm{dKL}(\delta|\hat{\delta}_j)}{\mathrm{dw}_1} = \sum \frac{\mathrm{dKL}(\delta|\hat{\delta}_j)}{\mathrm{d}\hat{\delta}_j}.\frac{\mathrm{d}\hat{\delta}_j}{\mathrm{dw}_1}}$$

This can be written as

$$\boxed{\sum \frac{\mathrm{dKL}(\delta|\hat{\delta}_j)}{\mathrm{dw}_1} = \frac{\mathrm{dKL}}{\mathrm{d}\underline{\hat{\delta}}}.\frac{\mathrm{d}\underline{\hat{\delta}}}{\mathrm{dw}_1}} where\ the\ (.)\ is\ element\ wise\ multiplication$$

This will be used as the partial derivative ok KL divergence term for gradient descent.

For the Tykhonov regularization term, the partial derivatives are as follows.

$$J_2 = \frac{\lambda}{2}(w_1 @ w_1{}^T + w_2 @ w_2{}^T)$$

$$\frac{\mathrm{d}J_2}{\mathrm{dw}_1} = \lambda. w_1$$

$$\frac{\mathrm{d}J_2}{\mathrm{dw}_2} = \lambda. w_2$$

The derivatives of the mean squared error loss is trivial as in the previous assignments and is:

$$J_1 = \frac{1}{2N}(x - Y)^T(x - Y) = \frac{1}{2N}(\|x - Y\|^2)$$

$$Y = \varphi(\varphi(x@\mathrm{w}_1 + b_1) + b2)$$

$$O_1 = \varphi(x@\mathrm{w}_1 + b_1)$$

$$delta2 = -\frac{1}{N}.(x - Y).\varphi'(\varphi(x@\mathrm{w}_1 + b_1) + b2)$$

$$\frac{\mathrm{d}J_2}{\mathrm{dw}_2} = O_1 @ delta2$$

$$delta1 = -(delta2@\mathrm{w}_2).\varphi'(x@\mathrm{w}_1 + b_1)$$

$$\frac{\mathrm{d}J_2}{\mathrm{dw}_1} = x@delta1$$

Similarly for biases

$$\frac{\mathrm{d}J_2}{\mathrm{db}_2} = \sum delta2$$

$$\frac{\mathrm{d}J_2}{\mathrm{db}_1} = \sum delta1$$

Abdul Basit Anees
21600659
EEE-443

The gradient descent is as follows:

```
#Gradient descent
w_1 = w_1 - rate * grad_w1
w_2 = w_2 - rate * grad_w2
b_1 = b_1 - rate * grad_b1
b_2 = b_2 - rate * grad_b2
```

The weights are initialized in the range [-w_0 w_0], where w_0 = $w_0 = \sqrt{\frac{6}{L_{pre}+L_{post}}} = 0.13693$

Lambda was chosen as 0.0005. After experimentation, Beta was chosen as 0.008 and learning rate of 0.01 gives the best loss curve. Initially beta was chosen as 10 which was causing all sigmoid outputs to go towards 0 or infinity. However, after choosing a significantly smaller beta, the network performed well. Batch size of 128 was used and the network was trained for a 1000 epochs

Following are the features extracted by the network by the first layer weights. i.e w_1.

C)

Rho = 0.2, 0.5 and 0.8 were tried and the following weights were obtained. (with beta 0.008)

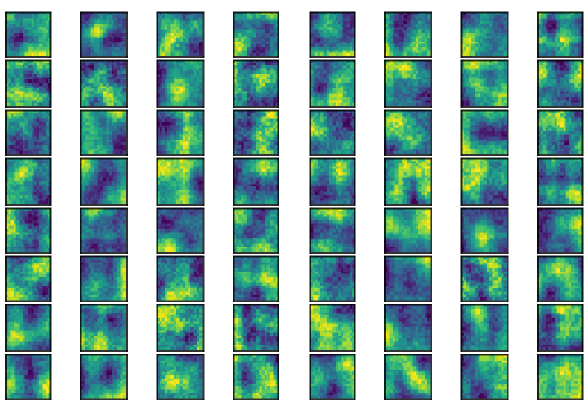
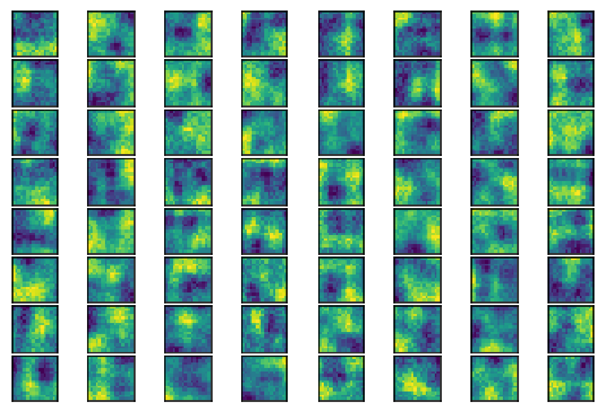


Figure 3: Rho = 0.25



Rho = 0.5

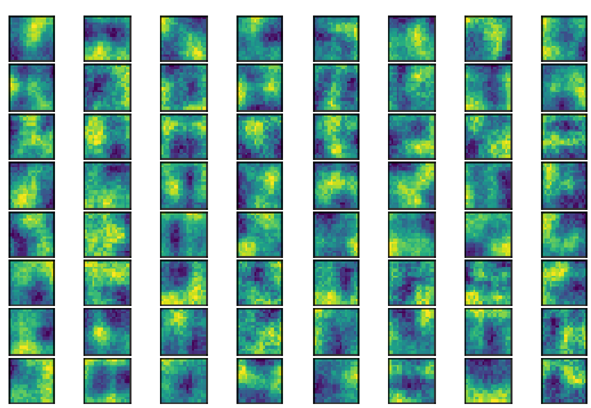


Figure 4: Rho = 0.8

Abdul Basit Anees
21600659
EEE-443

Rho = 0.25 was chosen since it gives more sparse features.

The features almost look like natural images. However, it can be seen that the features are not able to capture the high frequency information of the images but represent a compressed low frequency blurry representation of those images. The combination of several features is however able to reproduce the input image back to some extent as can be seen below.

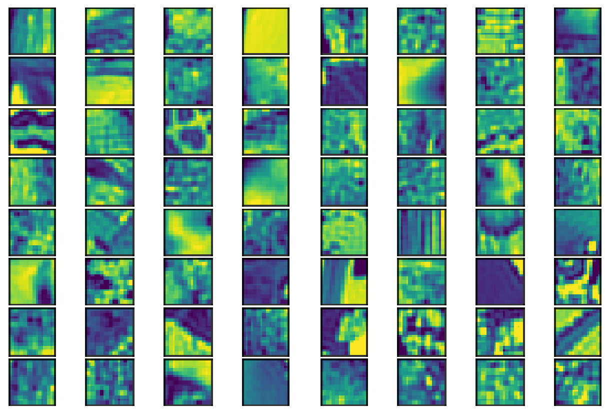


Figure 5: Input Images

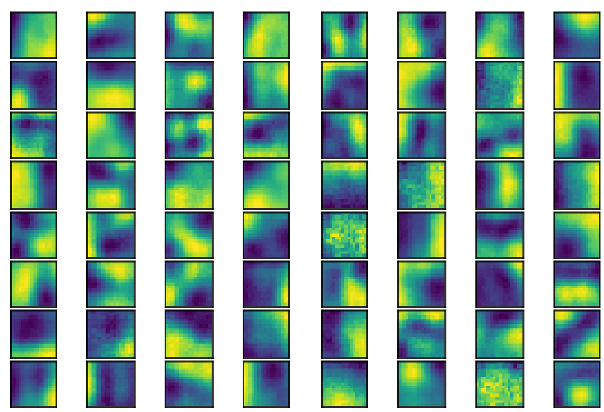
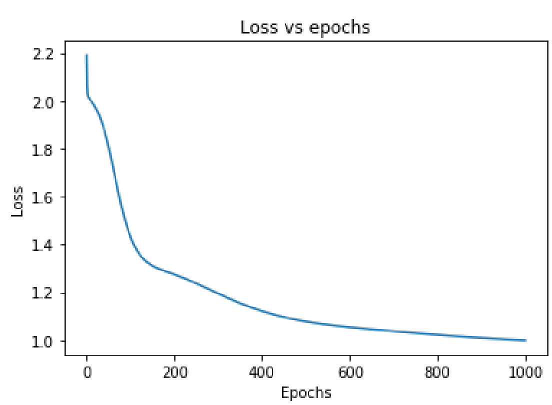


Figure 6: Decoded images after encoding

The loss goes down as follows:



Abdul Basit Anees
21600659
EEE-443

D

$\lambda = 0.001$ was chosen . A lower λ increases regularization to the extent that the model does not learn.

Beta = 0.008 and rho = 0.25

Three different values of encoding neurons were tried as L_hid = [25, 64, 100]

Following are the features extracted using the three different values:

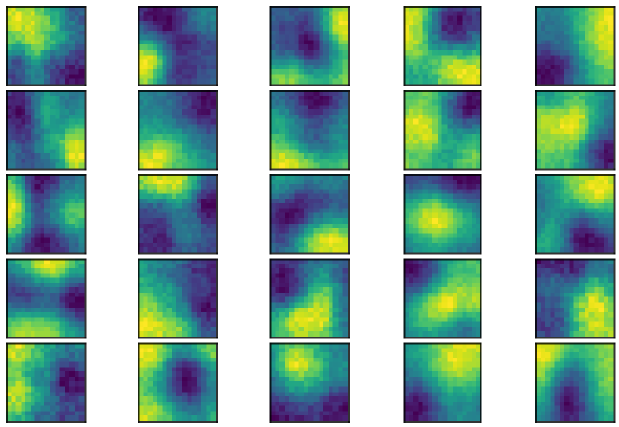


Figure 7: features for L_hid = 25

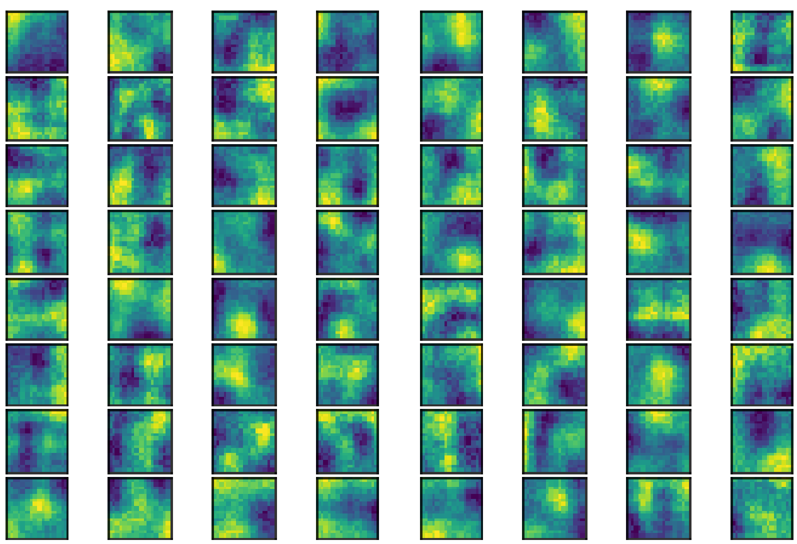


Figure 8: features for L_hid = 64

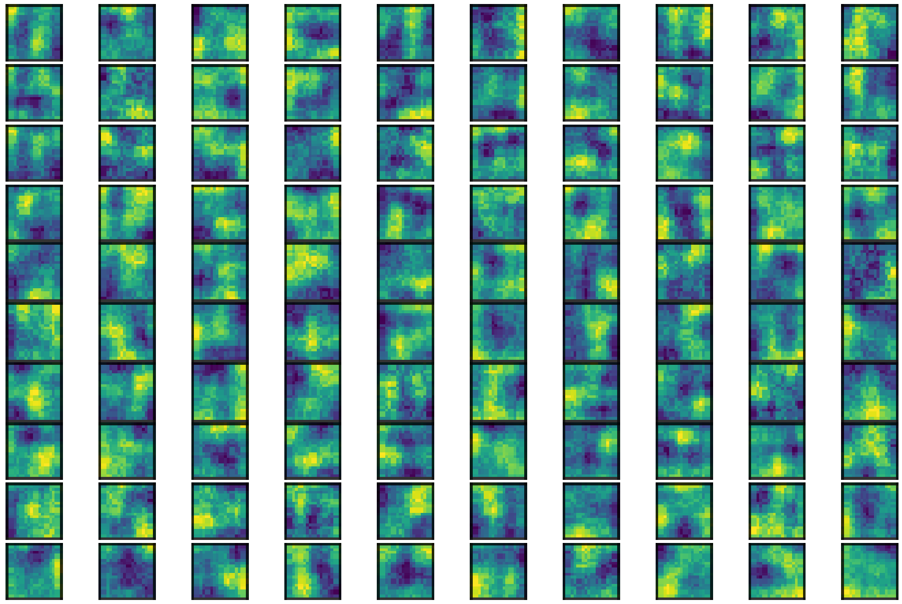


Figure 9: features for L_hid = 100

It can be seen that higher neurons lead to learning of more features that are also sparser. Lower number of neurons also decrease the scarcity of the features so as to generalize to the data better.

Then the decoded outputs are displayed corresponding to the following input:

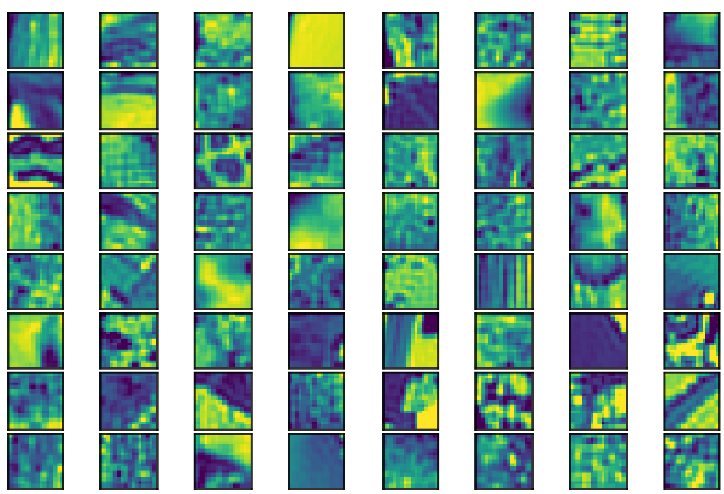


Figure 10: Input images

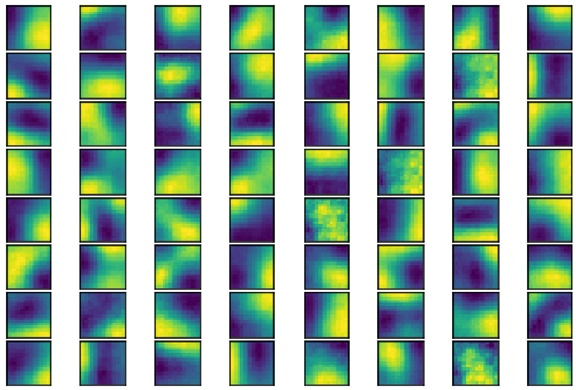
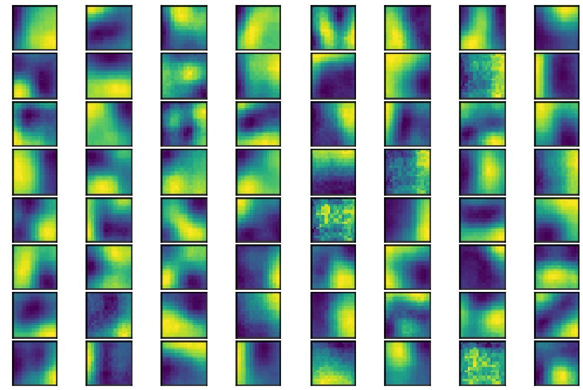


Figure 11: Output for L_hid = 25



Output for L_hid = 64

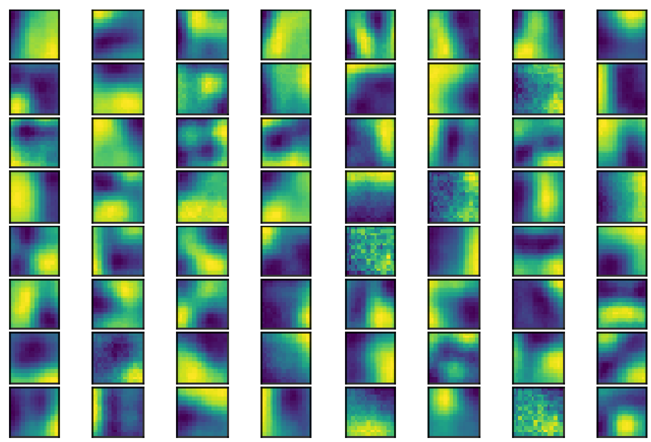


Figure 12: Output for L_hid = 100

In case of higher number of neurons, we can see that many of the generated outputs are sharper and closer to the actual image. This is because it has more feature detectors that can detect individual features different from the case of less neurons where many different features are generalized to a single neuron which causes the blurriness. This can also be seen from the feature weights displayed previously.

Then the loss across epochs is seen as:

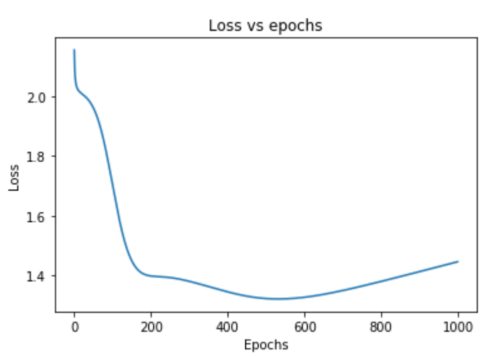
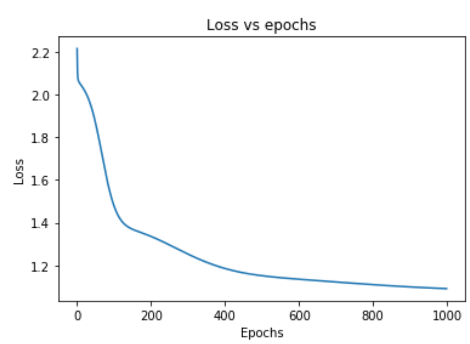


Figure 13: loss for L_hid = 25



loss for L_hid = 64

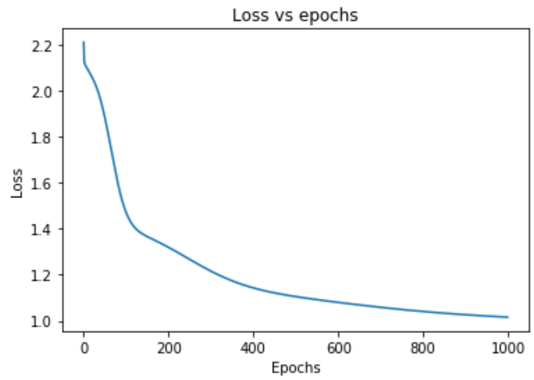


Figure 14: Loss for L_hid = 100

It can be seen that in case of lower number of neurons, the model trains quickly and then due to the comparatively higher learning rate for the model for 25 neurons, the error starts increasing again.

Question 2

Jupyter notebook attached at the end.

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
```

In [ ]:
```
naive cnn
```

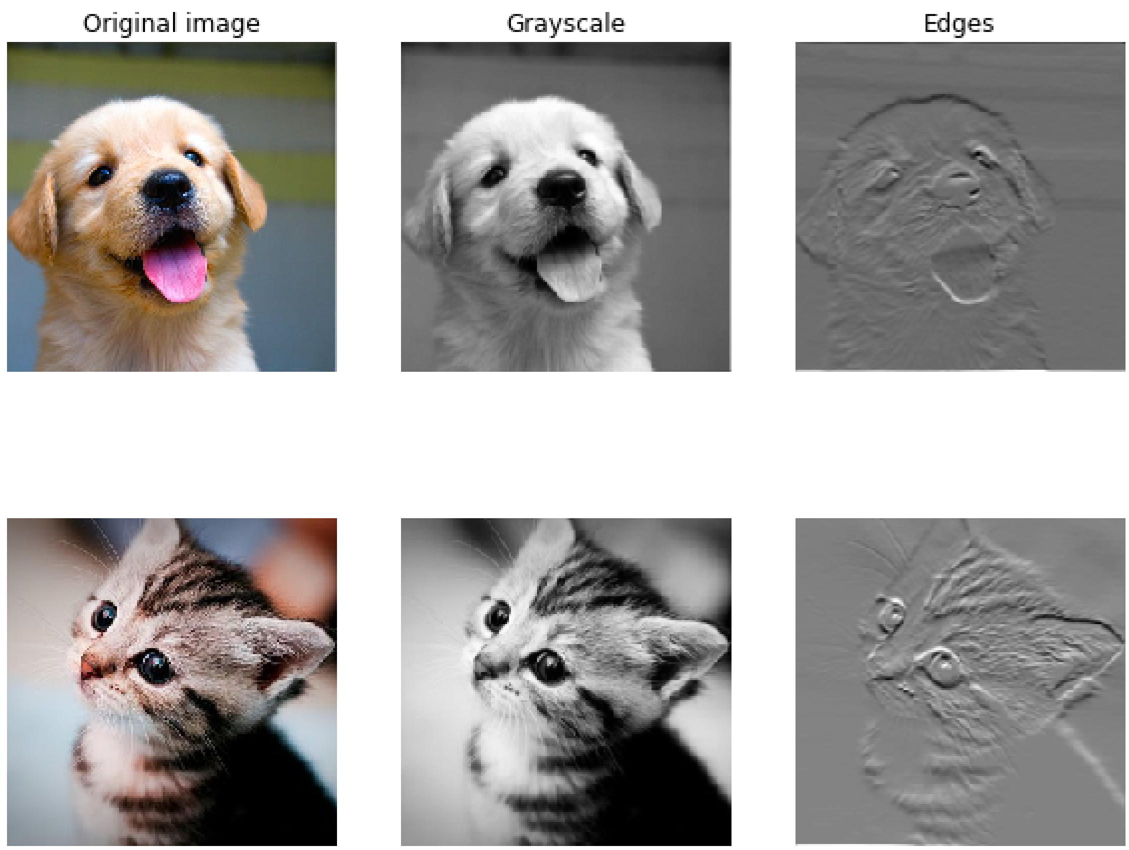
Aside: Image processing via convolutions

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

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

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

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

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

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

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

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

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

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

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

```
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```

```
/home/umair/.conda/envs/cs231n/lib/python3.7/site-packages/ipykernel_launche
r.py:3: DeprecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports u
ntil
/home/umair/.conda/envs/cs231n/lib/python3.7/site-packages/ipykernel_launche
r.py:10: DeprecationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # Remove the CWD from sys.path while we load stuff.
/home/umair/.conda/envs/cs231n/lib/python3.7/site-packages/ipykernel_launche
r.py:11: DeprecationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # This is added back by InteractiveShellApp.init_path()
```



Convolution: Naive backward pass

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

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

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

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

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

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

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

Max-Pooling: Naive forward

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

Check your implementation by running the following:

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

out, _ = max_pool_forward_naive(x, pool_param)

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

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

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

Max-Pooling: Naive backward

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

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

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

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

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

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

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

Fast layers

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

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

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

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

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

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

In [8]:

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

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

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

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

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

```
Testing conv_forward_fast:
Naive: 6.384810s
Fast: 0.010600s
Speedup: 602.335425x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 8.422491s
Fast: 0.016960s
Speedup: 496.597959x
dx difference:  1.949764775345631e-11
dw difference:  5.684079808685177e-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: 0.168703s
fast: 0.002104s
speedup: 80.180397x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.376410s
fast: 0.012205s
speedup: 30.840307x
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 [9]:
```
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 [10]:
```
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 [11]:
```
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 [12]:
```
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

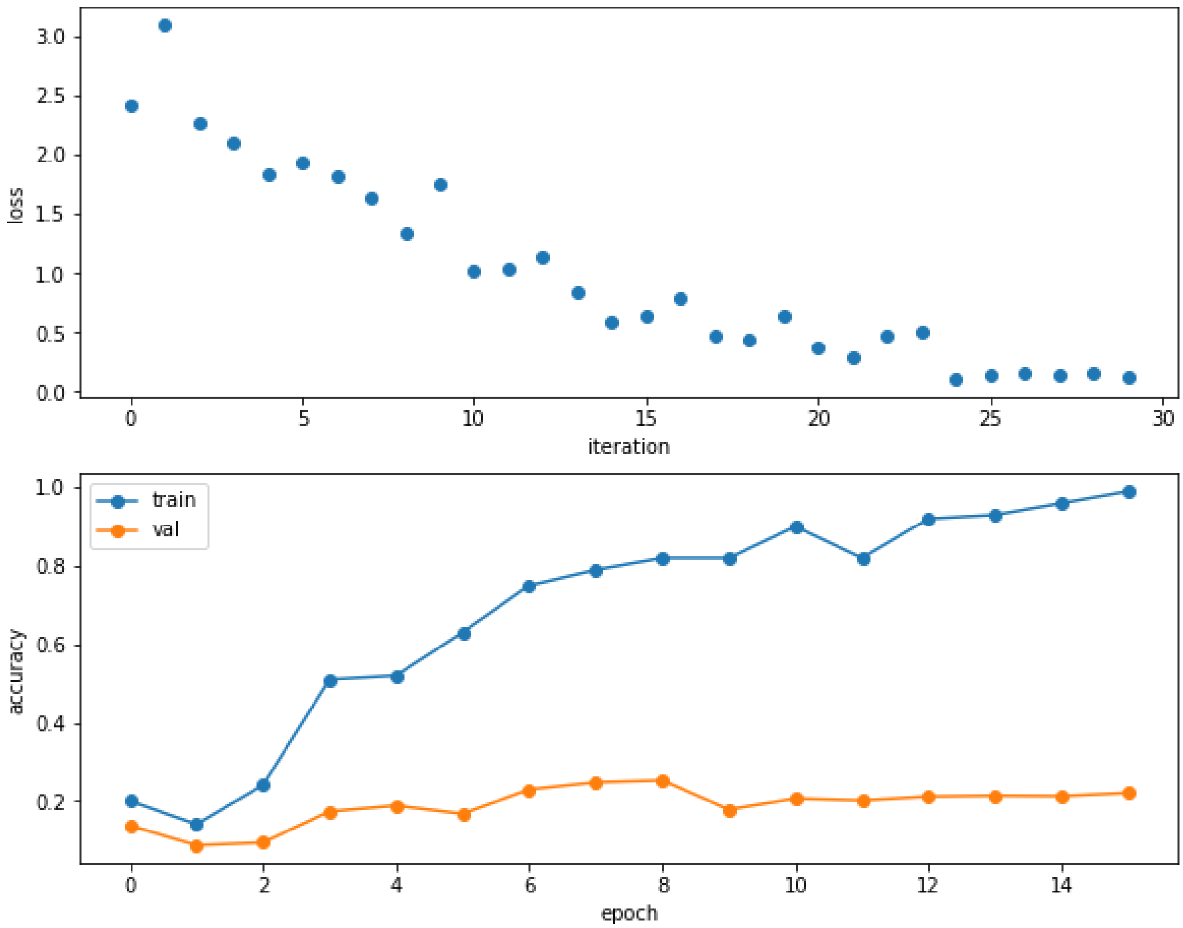
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

In [13]:
```
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 [14]:
```
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:

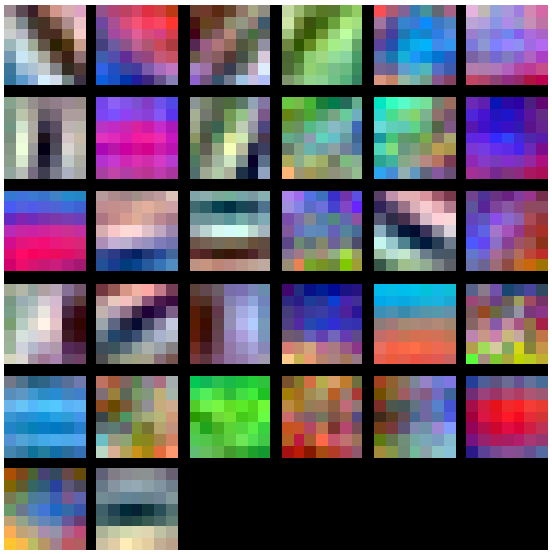
In [15]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

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 [16]:
```
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 [17]:
```
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 [18]:
```
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 [19]:
```
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 [20]:
```
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 [21]:
```
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
```

Comments

I created a separate environment for cs231n. After that I installed several requirements from the requirements.txt file. Then the Cython was built which is used for fast layers implementation of conv NNs. After that the code was ready to run. I ran the code and made observations along the way as can be seen in the output of each cell.

- Naive forward pass
- Filters for grayscale and edge detection are created and their outputs are visualized. As the name implies the grayscale filter converts the image to grayscale and the edge detection filters gives highlights the edges.
- Naive backward
- Max pooling naive forward
- Max pooling naive backward
- Fast layers implementation:
  - Without max pool: It can be seen the fast layers implementation is significantly faster than the naive one.
    - Testing conv_forward_fast: Speedup : 602.335425x
    - Testing conv_backward_fast: Speedup: 496.597959x
  - With max pool: Again faster
    - Testing pool_forward_fast: speedup : 80.180397x
    - Testing pool_backward_fast: speedup: 30.840307x
- Sandwich layers implementation
- Three layer conv net:
  - Sanity check loss: Loss is almost equal to ln(10) for 10 classes without regularization and increases with regularization
- Gradient check: The gradients are small and in the order of 10^-2 which is fine
- Overfit small data: Shows that the network works fine with 99% accuracy on training and 22% on validation data which shows overfitting which can be verified by the graphs too
- Train the network: The network achieves 50.2% accuracy on the training and 49.9% on the validation data after one epoch of training
- Visualize filters: It can be seen that the learnt filters have some patterns and can detect those in real iamge data
- Spatial batch normalization:
  - Before spatial batch normalization: (Arbitrary means and std)

    ```
    Means:  [9.33463814 8.90909116 9.11056338]
    Stds:  [3.61447857 3.19347686 3.5168142 ]
    ```

  - After spatial batch normalization: (0 mean and 1 std)

    ```
    Means:  [ 5.85642645e-16  5.93969318e-16 -8.88178420e-17]
    Stds:  [0.99999962 0.99999951 0.9999996 ]
    ```

  - After spatial batch normalization forward (nontrivial gamma, beta): Desired means and stds are obtained

    ```
    Means:  [6. 7. 8.]
    Stds:  [2.99999885 3.99999804 4.99999798]
    ```

- After spatial batch normalization forward (test-time): (means close to 0 and std close to 1)

  ```
  means:  [-0.08034406  0.07562881  0.05716371  0.04378383]
  stds:  [0.96718744 1.0299714  1.02887624 1.00585577]
  ```

- Gradient check for spatial batch normalization backward: The errors are close to zero

In [ ]:

What's this PyTorch business?

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

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

What is PyTorch?

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

Why?

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

PyTorch versions

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

How will I learn PyTorch?

Justin Johnson has made an excellent tutorial (https://github.com/jcjohnson/pytorch-examples) for PyTorch.

You can also find the detailed API doc (http://pytorch.org/docs/stable/index.html) here. If you have other questions that are not addressed by the API docs, the PyTorch forum (https://discuss.pytorch.org/) is a much better place to ask than StackOverflow.

Table of Contents

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

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

Here is a table of comparison:

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

Part I. Preparation

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

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

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

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

import numpy as np
```

In [2]:
```
NUM_TRAIN = 49000

# The torchvision.transforms package provides tools for preprocessing data
# and for performing data augmentation; here we set up a transform to
# preprocess the data by subtracting the mean RGB value and dividing by the
# standard deviation of each RGB value; we've hardcoded the mean and std.
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010
))
            ])

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

cifar10_val = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN, 5
0000)))

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

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

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

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

In [3]:
```
USE_GPU = True

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

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

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

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

```
using device: cuda
```

Part II. Barebones PyTorch

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

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

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

PyTorch Tensors: Flatten Function

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

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

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

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

In [4]:

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

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

test_flatten()
```

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


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

Barebones PyTorch: Two-Layer Network

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

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

In [5]:
```
import torch.nn.functional as F  # useful stateless functions

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

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

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

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

    w1, w2 = params

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


def two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature dimen
sion 50
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 10]

two_layer_fc_test()
```

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

Barebones PyTorch: Three-Layer ConvNet

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

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

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

In [6]:
```
def three_layer_convnet(x, params):
    """
    Performs the forward pass of a three-layer convolutional network with the
    architecture defined above.

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

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

    conv1 = F.conv2d(x, weight=conv_w1, bias=conv_b1, padding=2)
    relu1 = F.relu(conv1)
    conv2 = F.conv2d(relu1, weight=conv_w2, bias=conv_b2, padding=1)
    relu2 = F.relu(conv2)
    relu2_flat = flatten(relu2)
    scores = relu2_flat.mm(fc_w) + fc_b

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

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

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

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

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

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

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

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

Barebones PyTorch: Initialization

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

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

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

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

In [8]:
```
def random_weight(shape):
    """
    Create random Tensors for weights; setting requires_grad=True means that w
e
    want to compute gradients for these Tensors during the backward pass.
    We use Kaiming normalization: sqrt(2 / fan_in)
    """
    if len(shape) == 2:  # FC weight
        fan_in = shape[0]
    else:
        fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_channel, k
H, kW]
    # randn is standard normal distribution generator.
    w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fan_in)
    w.requires_grad = True
    return w

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

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

Out[8]:
```
tensor([[ 0.0569, -1.0300,  0.0631,  0.7757, -0.4663],
        [-0.1739,  0.0430,  0.6148, -0.4764, -0.0597],
        [-0.4246, -0.5188,  1.7372,  0.1569,  0.4756]], device='cuda:0',
       requires_grad=True)
```

Barebones PyTorch: Check Accuracy

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

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

In [9]:
```
def check_accuracy_part2(loader, model_fn, params):
    """
    Check the accuracy of a classification model.
    
    Inputs:
    - loader: A DataLoader for the data split we want to check
    - model_fn: A function that performs the forward pass of the model,
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the model
    
    Returns: Nothing, but prints the accuracy of the model
    """
    split = 'val' if loader.dataset.train else 'test'
    print('Checking accuracy on the %s set' % split)
    num_correct, num_samples = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.int64)
            scores = model_fn(x, params)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 
* acc))
```

BareBones PyTorch: Training Loop

We can now set up a basic training loop to train our network. We will train the model using stochastic gradient descent without momentum. We will use `torch.functional.cross_entropy` to compute the loss; you can [read about it here (http://pytorch.org/docs/stable/nn.html#cross-entropy)](http://pytorch.org/docs/stable/nn.html#cross-entropy).

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

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

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

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

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

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

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

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

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

BareBones PyTorch: Train a Two-Layer Network

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

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

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

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

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

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

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

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

```
Iteration 0, loss = 3.0076
Checking accuracy on the val set
Got 182 / 1000 correct (18.20%)

Iteration 100, loss = 2.3763
Checking accuracy on the val set
Got 308 / 1000 correct (30.80%)

Iteration 200, loss = 1.7362
Checking accuracy on the val set
Got 370 / 1000 correct (37.00%)

Iteration 300, loss = 2.1364
Checking accuracy on the val set
Got 366 / 1000 correct (36.60%)

Iteration 400, loss = 1.7328
Checking accuracy on the val set
Got 391 / 1000 correct (39.10%)

Iteration 500, loss = 1.5986
Checking accuracy on the val set
Got 413 / 1000 correct (41.30%)

Iteration 600, loss = 1.8272
Checking accuracy on the val set
Got 420 / 1000 correct (42.00%)

Iteration 700, loss = 1.5197
Checking accuracy on the val set
Got 472 / 1000 correct (47.20%)
```

BareBones PyTorch: Training a ConvNet

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

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

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

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

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

channel_1 = 32
channel_2 = 16

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

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

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

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

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

```
Iteration 0, loss = 2.9398
Checking accuracy on the val set
Got 118 / 1000 correct (11.80%)

Iteration 100, loss = 1.8403
Checking accuracy on the val set
Got 318 / 1000 correct (31.80%)

Iteration 200, loss = 1.4652
Checking accuracy on the val set
Got 418 / 1000 correct (41.80%)

Iteration 300, loss = 1.6245
Checking accuracy on the val set
Got 401 / 1000 correct (40.10%)

Iteration 400, loss = 1.5689
Checking accuracy on the val set
Got 429 / 1000 correct (42.90%)

Iteration 500, loss = 1.7237
Checking accuracy on the val set
Got 441 / 1000 correct (44.10%)

Iteration 600, loss = 1.7108
Checking accuracy on the val set
Got 468 / 1000 correct (46.80%)

Iteration 700, loss = 1.6144
Checking accuracy on the val set
Got 452 / 1000 correct (45.20%)
```

Part III. PyTorch Module API

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

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

To use the Module API, follow the steps below:

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

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

Module API: Two-Layer Network

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

In [11]:
```
class TwoLayerFC(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        # assign layer objects to class attributes
        self.fc1 = nn.Linear(input_size, hidden_size)
        # nn.init package contains convenient initialization methods
        # http://pytorch.org/docs/master/nn.html#torch-nn-init
        nn.init.kaiming_normal_(self.fc1.weight)
        self.fc2 = nn.Linear(hidden_size, num_classes)
        nn.init.kaiming_normal_(self.fc2.weight)

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

def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64, featu
re dimension 50
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_TwoLayerFC()
```

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

Module API: Three-Layer ConvNet

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

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

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

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

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

In [12]:
```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        ########################################################################
##
        # TODO: Set up the layers you need for a three-layer ConvNet with the
#
        # architecture defined above.
#
        ########################################################################
##

        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2
, bias=True)
        nn.init.kaiming_normal_(self.conv1.weight)
        nn.init.constant_(self.conv1.bias, 0)

        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1,
bias=True)
        nn.init.kaiming_normal_(self.conv2.weight)
        nn.init.constant_(self.conv2.bias, 0)

        self.fc = nn.Linear(channel_2*32*32, num_classes)
        nn.init.kaiming_normal_(self.fc.weight)
        nn.init.constant_(self.fc.bias, 0)

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

    def forward(self, x):
        scores = None
        ########################################################################
##
        # TODO: Implement the forward function for a 3-layer ConvNet. you
#
        # should use the layers you defined in __init__ and specify the
#
        # connectivity of those layers in forward()
#
        ########################################################################
##

        relu1 = F.relu(self.conv1(x))
        relu2 = F.relu(self.conv2(relu1))
        scores = self.fc(flatten(relu2))

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

```
        return scores


def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image
 size [3, 32, 32]
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8, num_cla
sses=10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_ThreeLayerConvNet()
```

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

Module API: Check Accuracy

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

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

In [13]:

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

Module API: Training Loop

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

In [14]:
```
def train_part34(model, optimizer, epochs=1):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

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

    Returns: Nothing, but prints model accuracies during training.
    """
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

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

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

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

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

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

Module API: Train a Two-Layer Network

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

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

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

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

In [17]:
```
hidden_layer_size = 4000
learning_rate = 1e-2
model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.1124
Checking accuracy on validation set
Got 135 / 1000 correct (13.50)

Iteration 100, loss = 2.2389
Checking accuracy on validation set
Got 300 / 1000 correct (30.00)

Iteration 200, loss = 1.8365
Checking accuracy on validation set
Got 369 / 1000 correct (36.90)

Iteration 300, loss = 2.0818
Checking accuracy on validation set
Got 381 / 1000 correct (38.10)

Iteration 400, loss = 1.4971
Checking accuracy on validation set
Got 410 / 1000 correct (41.00)

Iteration 500, loss = 1.6927
Checking accuracy on validation set
Got 431 / 1000 correct (43.10)

Iteration 600, loss = 1.3435
Checking accuracy on validation set
Got 429 / 1000 correct (42.90)

Iteration 700, loss = 1.9418
Checking accuracy on validation set
Got 418 / 1000 correct (41.80)
```

Module API: Train a Three-Layer ConvNet

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

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

In [18]:
```
learning_rate = 3e-3
channel_1 = 32
channel_2 = 16

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

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

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.8775
Checking accuracy on validation set
Got 118 / 1000 correct (11.80)

Iteration 100, loss = 1.7695
Checking accuracy on validation set
Got 345 / 1000 correct (34.50)

Iteration 200, loss = 1.6995
Checking accuracy on validation set
Got 393 / 1000 correct (39.30)

Iteration 300, loss = 1.8308
Checking accuracy on validation set
Got 425 / 1000 correct (42.50)

Iteration 400, loss = 1.6003
Checking accuracy on validation set
Got 441 / 1000 correct (44.10)

Iteration 500, loss = 1.4502
Checking accuracy on validation set
Got 479 / 1000 correct (47.90)

Iteration 600, loss = 1.4112
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)

Iteration 700, loss = 1.3603
Checking accuracy on validation set
Got 503 / 1000 correct (50.30)
```

Part IV. PyTorch Sequential API

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

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

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

Sequential API: Two-Layer Network

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

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

In [15]:
```python
# We need to wrap `flatten` function in a module in order to stack it
# in nn.Sequential
class Flatten(nn.Module):
    def forward(self, x):
        return flatten(x)

hidden_layer_size = 4000
learning_rate = 1e-2

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

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.2830
Checking accuracy on validation set
Got 170 / 1000 correct (17.00)

Iteration 100, loss = 1.4929
Checking accuracy on validation set
Got 393 / 1000 correct (39.30)

Iteration 200, loss = 1.7012
Checking accuracy on validation set
Got 403 / 1000 correct (40.30)

Iteration 300, loss = 1.8348
Checking accuracy on validation set
Got 414 / 1000 correct (41.40)

Iteration 400, loss = 1.8167
Checking accuracy on validation set
Got 422 / 1000 correct (42.20)

Iteration 500, loss = 1.8693
Checking accuracy on validation set
Got 444 / 1000 correct (44.40)

Iteration 600, loss = 1.6211
Checking accuracy on validation set
Got 441 / 1000 correct (44.10)

Iteration 700, loss = 1.5587
Checking accuracy on validation set
Got 410 / 1000 correct (41.00)
```

Sequential API: Three-Layer ConvNet

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

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

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

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

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

In [16]:
```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
optimizer = None

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

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

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

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

model.apply(init_weights)

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3069
Checking accuracy on validation set
Got 152 / 1000 correct (15.20)

Iteration 100, loss = 1.5757
Checking accuracy on validation set
Got 438 / 1000 correct (43.80)

Iteration 200, loss = 1.4451
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)

Iteration 300, loss = 1.3991
Checking accuracy on validation set
Got 541 / 1000 correct (54.10)

Iteration 400, loss = 1.5362
Checking accuracy on validation set
Got 541 / 1000 correct (54.10)

Iteration 500, loss = 1.1519
Checking accuracy on validation set
Got 581 / 1000 correct (58.10)

Iteration 600, loss = 1.1274
Checking accuracy on validation set
Got 573 / 1000 correct (57.30)

Iteration 700, loss = 1.1996
Checking accuracy on validation set
Got 589 / 1000 correct (58.90)
```

Part V. CIFAR-10 open-ended challenge

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

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

Describe what you did at the end of this notebook.

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

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

Things you might try:

- **Filter size**: Above we used 5x5; would smaller filters be more efficient?
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?
- **Batch normalization**: Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster?
- **Network architecture**: The network above has two layers of trainable parameters. Can you do better with a deep network? Good architectures to try include:
  - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling**: Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1 image picture (1, 1 , Filter#), which is then reshaped into a (Filter#) vector. This is used in [Google's Inception Network (https://arxiv.org/abs/1512.00567)](https://arxiv.org/abs/1512.00567) (See Table 1 for their architecture).
- **Regularization**: Add l2 weight regularization, or perhaps use Dropout.

Tips for training

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

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.

- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.
- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

Going above and beyond

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

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.
- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.
- Model ensembles
- Data augmentation
- New Architectures
  - ResNets (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)

In [21]:
```
###############################################################################
##
# TODO:
#
# Experiment with any architectures, optimizers, and hyperparameters.
#
# Achieve AT LEAST 70% accuracy on the *validation set* within 10 epochs.
#
#
#
# Note that you can use the check_accuracy function to evaluate on either
#
# the test set or the validation set, by passing either loader_test or
#
# loader_val as the second argument to check_accuracy. You should not touch
#
# the test set until you have finished your architecture and  hyperparameter
#
# tuning, and only run the test set once at the end to report a final value.
#
###############################################################################
##
model = None
optimizer = None

# A 4-layer convolutional network
# (conv -> batchnorm -> relu -> maxpool) * 3 -> fc
layer1 = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=5, padding=2),
    nn.BatchNorm2d(16),
    nn.ReLU(),
    nn.MaxPool2d(2)
)

layer2 = nn.Sequential(
    nn.Conv2d(16, 32, kernel_size=3, padding=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d(2)
)

layer3 = nn.Sequential(
    nn.Conv2d(32, 64, kernel_size=3, padding=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(2)
)

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

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

```
)

learning_rate = 1e-3

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

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

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

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

```
Iteration 0, loss = 2.6101
Checking accuracy on validation set
Got 154 / 1000 correct (15.40)

Iteration 0, loss = 0.7711
Checking accuracy on validation set
Got 645 / 1000 correct (64.50)

Iteration 0, loss = 0.8233
Checking accuracy on validation set
Got 689 / 1000 correct (68.90)

Iteration 0, loss = 0.8043
Checking accuracy on validation set
Got 725 / 1000 correct (72.50)

Iteration 0, loss = 0.6599
Checking accuracy on validation set
Got 739 / 1000 correct (73.90)

Iteration 0, loss = 0.6278
Checking accuracy on validation set
Got 743 / 1000 correct (74.30)

Iteration 0, loss = 0.6053
Checking accuracy on validation set
Got 735 / 1000 correct (73.50)

Iteration 0, loss = 0.4958
Checking accuracy on validation set
Got 747 / 1000 correct (74.70)

Iteration 0, loss = 0.7287
Checking accuracy on validation set
Got 740 / 1000 correct (74.00)

Iteration 0, loss = 0.4133
Checking accuracy on validation set
Got 732 / 1000 correct (73.20)
```

Describe what you did

In the cell below you should write an explanation of what you did, any additional features that you implemented, and/or any graphs that you made in the process of training and evaluating your network.

Firstly, I installed cuda for gpu computing for pytorch. Then I created a separate environment for cs231n. After that I installed several requirements from the requirements.txt file. After that the code was ready to run. I ran the code and made observations along the way as can be seen in the output of each cell.

- Downloaded CIFAR10 dataset inputting the data to a data loader which will load the data in batches instead of using them all at once.
- Flatten tensors Barebones NN:
- Write fuctions for fowards pass for 2 layers and conv net for 3 layers
- Write function for training and accuracy
- Train and test 2 layer network: Accuracy passses the required threshold: Val Accuracy= 47.2%
- Train and test 3 layer conv net: Accuracy passses the required threshold: Val Accuracy= 45.2%

Module API:

- Write fuctions for fowards pass for 2 layers and conv net for 3 layers
- Write function for training and accuracy
- Train and test 2 layer network: Accuracy passses the required threshold: Val Accuracy= 41.8%
- Train and test 3 layer conv net: Accuracy passses the required threshold: Val Accuracy= 50.3%

Sequential API:

- Write fuctions for fowards pass for 2 layers and conv net for 3 layers
- Write function for training and accuracy
- Train and test 2 layer network: Accuracy passses the required threshold: Val Accuracy= 43.9%
- Train and test 3 layer conv net: Accuracy passses the required threshold: Val Accuracy= 57.6%

Sequential Conv net performs best, then Module Conv net and barebones convnet the worst

CIFAR 10 Challenge: Val accuracy of 73.9% is achieved.

- We use small (3,3) filters and it can be seen that they are more efficient than (5,5) filters
- We use lesser filters (16) in the first layer and keep increasing number of filters (32, 64) until the final layer and they perform better
- Max pooling works better for generalizing the data better so it perform better on the test data and also reduces the data.
- A deeper network will be able to learn the model better but will require more training time

Test set -- run this only once

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

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

```
Checking accuracy on test set
Got 7401 / 10000 correct (74.01)
```

In [ ]:

```

```

Abdul Basit Anees
21600659
EEE-443

Appendix

```
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 17 21:56:22 2019

@author: basit
"""

import sys
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import scipy.io
import pylab as py
import h5py
import random

question = '1'

def AbdulBasit_Anees_21600659_hw1(question):
    if question == '1' :
        print(question)
        return Q2()

def Q2():
    print("Running Q2a")
    data_norm = Q2a()
    print('Running Q2 b and c')
    Q2bc(data_norm)
```

Abdul Basit Anees
21600659
EEE-443

```
    print("Running Q2d")
    Q2d(data_norm)

def Q2d(data_norm):
    x_tr1 = data_norm.reshape((data_norm.shape[0], 16 * 16))

    #Shuffle data
    arr   = np.arange(x_tr1.shape[0])
    np.random.shuffle(arr)
    #x_tr  = x_tr[arr]
    x_tr1 = x_tr1[arr]

    #Initialize network parameters
    lambd      = 0.001 #0:0.001
    beta       = 0.008
    rho        = 0.25
    epochs     = 1000
    batch_size = 128
    rate       = 0.01
    nBatches   = int(np.ceil(x_tr1.shape[0]/batch_size))
    J_train    = np.zeros((epochs, 1))

    mse_tr  = {}
    weights = {}
    N = [25, 64,100]
    nrows = [5, 8, 10]
    for L_hid in N:
        w0        = np.sqrt(6/(L_hid+256))
        w1        = np.random.uniform(-w0, w0, (256, L_hid))
        w2        = np.random.uniform(-w0, w0, (L_hid, 256))
```

```
    b1        = np.random.uniform(-w0, w0, (1, L_hid))
    b2        = np.random.uniform(-w0, w0, (1, 256))
    for nEpoch in range(epochs):
        w1, w2, b1, b2    = train (w1, w2, b1 , b2, x_tr1, x_tr1, batch_size, rate, nBatches, rho, lambd, beta)
        yp_tr             = forward(w1, w2, b1 , b2, x_tr1)
        mse_cost = (0.5 / x_tr1.shape[0]) * np.sum((yp_tr-x_tr1)**2)
        tykhonov = (lambd / 2) * (np.sum(w1**2) + np.sum(w2**2))
        rhos = (1 / x_tr1.shape[0]) * np.sum(sigmoid(x_tr1 @ w1), axis = 0)
        kl_diver = beta * KL_div(rho, rhos)
        J_train[nEpoch]   = mse_cost + tykhonov + kl_diver
        if nEpoch%50 == 0:
            print(nEpoch)
        if nEpoch%100 == 0:
            print(J_train[nEpoch])

    mse_tr[str(L_hid)]  = np.copy(J_train)
    weights[str(L_hid)] = {'w1': np.copy(w1), 'w2': np.copy(w2), 'b1': np.copy(b1), 'b2': np.copy(b2)}

for i in range(len(N)):
    weights1 = ((weights[str(N[i])])['w1']).reshape((16,16,N[i]))
    make_grid1(weights1, nrows[i], nrows[i])

make_grid1(data_norm[:64].transpose((1,2,0)), 8, 8)

wn1 = weights[str(N[0])]
wn2 = weights[str(N[1])]
wn3 = weights[str(N[2])]
wn = [wn1, wn2, wn3]
#wn = [wn1]
```

```
    for i in range(len(N)):
        out = forward((wn[i])['w1'], (wn[i])['w2'], (wn[i])['b1'] , (wn[i])['b2'], data_norm[:64].reshape((64,256)))
        make_grid1(out.reshape(64,16,16).transpose((1,2,0)), 8, 8)

    for i in range(len(N)):
        loss = mse_tr[str(N[i])]
        plt.figure()
        plt.plot(np.arange(len(loss)), loss)
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.title('Loss vs epochs')

def Q2bc(data_norm):
    x_tr1 = data_norm.reshape((data_norm.shape[0], 16 * 16))

    #Shuffle data
    arr   = np.arange(x_tr1.shape[0])
    np.random.shuffle(arr)
    x_tr1 = x_tr1[arr]

    #Initialize network parameters
    L_hid      = 64 #10:100
    lambd      = 0.0005 #0:0.001
    beta       = 0.0002
    rho        = 0.8
    epochs     = 1000
    batch_size = 128
    rate       = 0.01
    w0         = np.sqrt(6/(L_hid+256))
    nBatches   = int(np.ceil(x_tr1.shape[0]/batch_size))
```

```
w1        = np.random.uniform(-w0, w0, (256, L_hid))
w2        = np.random.uniform(-w0, w0, (L_hid, 256))
b1        = np.random.uniform(-w0, w0, (1, L_hid))
b2        = np.random.uniform(-w0, w0, (1, 256))
J_train    = np.zeros((epochs, 1))

#    mse_cost = (0.5 / x_tr1.shape[0]) * np.sum((forward(w1, w2, b1 , b2, x_tr1)-x_tr1)**2)
#    tykhonov = (lambd / 2) * (np.sum(w1**2) + np.sum(w2**2))
#    rhos = (1 / x_tr1.shape[0]) * np.sum(sigmoid(x_tr1 @ w1), axis = 0)
#    kl_diver = beta * KL_div(rho, rhos)
#    cost = mse_cost + tykhonov + kl_diver

for nEpoch in range(epochs):
    w1, w2, b1, b2    = train (w1, w2, b1 , b2, x_tr1, x_tr1, batch_size, rate, nBatches, rho, lambd, beta)
    yp_tr            = forward(w1, w2, b1 , b2, x_tr1)
    mse_cost = (0.5 / x_tr1.shape[0]) * np.sum((yp_tr-x_tr1)**2)
    tykhonov = (lambd / 2) * (np.sum(w1**2) + np.sum(w2**2))
    rhos = (1 / x_tr1.shape[0]) * np.sum(sigmoid(x_tr1 @ w1), axis = 0)
    kl_diver = beta * KL_div(rho, rhos)
    J_train[nEpoch]   = mse_cost + tykhonov + kl_diver
    if nEpoch%50 == 0:
        print(nEpoch)
    if nEpoch%25 == 0:
        print(J_train[nEpoch])

weights1 = w1.reshape((16,16,64))

def make_grid1(weights, rows, col):
#    col = 8
#    rows = 8
```

```
        _, grid = plt.subplots(nrows=rows,ncols=col)
        plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace= None, hspace=0.08)
        for p in range(rows*col):
            grid[int(p/col),p%col].imshow(weights[:,:,p])     #, cmap = 'gray'
            grid[int(p/col),p%col].axes.get_xaxis().set_visible(False)
            grid[int(p/col),p%col].axes.get_yaxis().set_visible(False)

    make_grid1(weights1, 8, 8)


    plt.figure()
    make_grid1(data_norm[:64].transpose((1,2,0)), 8, 8)
    out = forward(w1, w2, b1 , b2, data_norm[:64].reshape((64,256)))
    plt.figure()
    make_grid1(out.reshape(64,16,16).transpose((1,2,0)), 8, 8)


    plt.figure()
    plt.plot(np.arange(len(J_train)), J_train)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Loss vs epochs')


def Q2a():
    #Load data
    with h5py.File("C:/4th Year/EEE443/HW3/assign3_data1.h5", 'r') as f:
        keys = list(f.keys())
        data = f[keys[0]].value
#       invXForm = f[keys[1]].value
#       xForm = f[keys[2]].value
        f.close()
```

Abdul Basit Anees
21600659
EEE-443

```
# Y = 0.2126* R + 0.7152*G + 0.0722*B
scale = np.array((0.2126, 0.7152, 0.0722)).reshape((3,1,1))

data_norm = np.zeros((len(data),16,16))
for i in range(len(data)):
    iData = np.sum(data[i]*scale, axis = 0)
    iData -= iData.mean()
    data_norm[i] = iData

clip = 3 * data_norm.std()
data_norm[(data_norm > clip)] = clip
data_norm[(data_norm < -clip)] = -clip
data_norm = ((data_norm + clip) * (0.8 / (2 * clip))) + 0.1

def make_grid(images):
    col = 17
    rows = 12
    for i in range(1):
        _, grid = plt.subplots(nrows=rows,ncols=col)
        plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace= None, hspace=0.08)
        plt.tight_layout()
        for p in range(rows*col):
            if(len(images.shape) == 4):
                image = images[p+(i*col*rows)]
                image = image - image.min()
                image = (image / image.max())
                grid[int(p/col),p%col].imshow(image.transpose((1,2,0)).reshape((16,16,3)))
                save = "fig"+ str(i)
            else:
                grid[int(p/col),p%col].imshow(images[p+(i*col*rows)])
```

Abdul Basit Anees
21600659
EEE-443

```
            save = "fig"+ str(i+1)
          grid[int(p/col),p%col].axes.get_xaxis().set_visible(False)
          grid[int(p/col),p%col].axes.get_yaxis().set_visible(False)
        plt.savefig(save + ".png", dpi = 600)

    index = np.arange(10240)
    random.shuffle(index)
    make_grid(data[index])
    make_grid(data_norm[index])
    return data_norm

#Plotting functions
def make_grid1(weights, rows, col):
#    col = 8
#    rows = 8
    plt.figure()
    _, grid = plt.subplots(nrows=rows,ncols=col)
    plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace= None, hspace=0.08)
    for p in range(rows*col):
        grid[int(p/col),p%col].imshow(weights[:,:,p])     #, cmap = 'gray'
        grid[int(p/col),p%col].axes.get_xaxis().set_visible(False)
        grid[int(p/col),p%col].axes.get_yaxis().set_visible(False)

#Functions
def tanh(x):
    return (np.exp(2*x) - 1) / (np.exp(2*x) + 1)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Abdul Basit Anees
21600659
EEE-443

```
def sigmoidGradient(x):
    return sigmoid(x)*(1-sigmoid(x))


def Relu(x):
    return x*(x>0)


def ReluGradient(x):
    return x>0


def forward(w_1, w_2, b_1, b_2, x_tr):
    o_1  = sigmoid((x_tr @ w_1) + b_1)
    y_p  = sigmoid((o_1 @ w_2) + b_2)
    return y_p


# calculate the kl divergence
def KL_div(rho, q):
    kl = rho*np.log(rho/q) + (1-rho)*np.log((1-rho)/(1-q))
    kl[np.isnan(kl)] = 0
    return ((1 / kl.shape[0]) * np.sum(kl))


def KL_der(rho, q):
    return ((1-rho)/(1-q)) - (rho/q)


#Training including forward and backward pass for one epoch
def train(w_1, w_2, b_1, b_2, x, y, batch_size, rate, nBatches, rho, lambd, beta):
    for i in range(nBatches):
        #Load batch
        batch_x = x[i*batch_size:(i+1)*batch_size]
        batch_y = y[i*batch_size:(i+1)*batch_size]
        #Forward pass
```

Abdul Basit Anees
21600659
EEE-443

```
        o_1  = sigmoid((batch_x @ w_1) + b_1)
#       o_11 = np.hstack((np.ones((o_1.shape[0],1)), o_1))
        y_p  = sigmoid((o_1 @ w_2) + b_2)
        #Backward pass
        delta_2 = - (1 / batch_size) * (batch_y - y_p) * sigmoidGradient(y_p)
        grad_w2 = o_1.T @ delta_2 + (lambd * w_2)
        grad_b2 = np.sum(delta_2, axis = 0) #/ delta_2.shape[0]

        rhos    = (1 / o_1.shape[0]) * np.sum(o_1, axis = 0)
        rho_der = KL_der(rho, rhos).reshape((1,rhos.shape[0]))
        rho_der_w1 = (1/batch_x.shape[0]) * (batch_x.T @ sigmoidGradient(o_1))
        rho_der_w  = rho_der_w1 * rho_der
        rho_der_b1 = (1/batch_x.shape[0]) * (np.sum(sigmoidGradient(o_1), axis = 0))
        rho_der_b  = rho_der_b1 * rho_der

        delta_1 = (delta_2 @ w_2.T) * sigmoidGradient(o_1)
        grad_w1 = batch_x.T @ delta_1 + (lambd * w_1) + (beta * rho_der_w)
        grad_b1 = (np.sum(delta_1, axis = 0) ) + (beta * rho_der_b)
        #Gradient descent
        w_1 = w_1 - rate * grad_w1
        w_2 = w_2 - rate * grad_w2
        b_1 = b_1 - rate * grad_b1
        b_2 = b_2 - rate * grad_b2
    return w_1, w_2 , b_1, b_2

output = AbdulBasit_Anees_21600659_hw1(question)
```